**NAVAL POSTGRADUATE SCHOOL**

**MONTEREY, CALIFORNIA**

**THESIS**

# CROSS-PLATFORM MOBILE APPLICATION DEVELOPMENT: A PATTERN-BASED APPROACH

by

Christian G. Acord
Corey C. Murphy

March 2012

| | |
|---|---|
| Thesis Advisor: | Thomas Otani |
| Co-Advisor: | John Gibson |

| REPORT DOCUMENTATION PAGE | | | *Form Approved OMB No. 0704-0188* |
|---|---|---|---|
| Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503. | | | |
| **1. AGENCY USE ONLY *(Leave blank)*** | **2. REPORT DATE** March 2012 | **3. REPORT TYPE AND DATES COVERED** Master's Thesis | |
| **4. TITLE AND SUBTITLE** Cross-Platform Mobile Application Development: A Pattern-Based Approach | | | **5. FUNDING NUMBERS** |
| **6. AUTHOR(S)** Christian G. Acord and Corey C. Murphy | | | |
| **7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Naval Postgraduate School Monterey, CA 93943-5000 | | | **8. PERFORMING ORGANIZATION REPORT NUMBER** |
| **9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)** N/A | | | **10. SPONSORING/MONITORING AGENCY REPORT NUMBER** |
| **11. SUPPLEMENTARY NOTES** The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ______N/A__________. | | | |
| **12a. DISTRIBUTION / AVAILABILITY STATEMENT** Approved for public release; distribution is unlimited | | | **12b. DISTRIBUTION CODE** |

**13. ABSTRACT (maximum 200 words)**

Mobile devices are fast becoming ubiquitous in today's society. New devices are constantly being released with unique combinations of hardware and software, or platforms. In order to support the ever-increasing number of platforms, developers must embrace some method of cross-platform development in which the design and implementation of applications for different platforms may be streamlined.

This thesis compares and contrasts two platforms, iOS and Android smartphones, and discusses how one might apply the Model, View, Controller pattern in order to minimize the inherent differences between the platforms. Furthermore, this thesis describes the Unified Design Process that can be used to implement native iOS and Android applications from a single design process. This design process reduces the amount of time required for the development of applications and maintains platform specific UI styles for the different platforms.

The authors used this process to design and build a functional prototype of the NPS Muster application on both platforms. This application is capable of displaying announcements and allowing NPS students to conduct daily musters.

| | | | |
|---|---|---|---|
| **14. SUBJECT TERMS** Mobile development, Cross-platform development, Android development, iPhone development | | | **15. NUMBER OF PAGES** 157 |
| | | | **16. PRICE CODE** |
| **17. SECURITY CLASSIFICATION OF REPORT** Unclassified | **18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified | **19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified | **20. LIMITATION OF ABSTRACT** UU |

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

# CROSS-PLATFORM MOBILE APPLICATION DEVELOPMENT: A PATTERN-BASED APPROACH

Christian G. Acord
Lieutenant, United States Navy
B.S., United States Naval Academy, 2006

Corey C. Murphy
Lieutenant, United States Navy
B.S., Duquesne University, 2006

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER SCIENCE IN COMPUTER SCIENCE**

from the

**NAVAL POSTGRADUATE SCHOOL**
**March 2012**

| | |
|---|---|
| Authors: | Christian G. Acord<br>Corey C. Murphy |
| Approved by: | Thomas Otani<br>Thesis Advisor |
| | John Gibson<br>Thesis Co-Advisor |
| | Dr. Peter J. Denning<br>Chair, Department of Computer Science |

THIS PAGE INTENTIONALLY LEFT BLANK

## ABSTRACT

Mobile devices are fast becoming ubiquitous in today’s society. New devices are constantly being released with unique combinations of hardware and software, or platforms. In order to support the ever-increasing number of platforms, developers must embrace some method of cross-platform development in which the design and implementation of applications for different platforms may be streamlined.

This thesis compares and contrasts two platforms, iOS and Android smartphones, and discusses how one might apply the Model, View, Controller pattern in order to minimize the inherent differences between the platforms. Furthermore, this thesis describes the Unified Design Process that can be used to implement native iOS and Android applications from a single design process. This design process reduces the amount of time required for the development of applications and maintains platform specific UI styles for the different platforms.

The authors used this process to design and build a functional prototype of the NPS Muster application on both platforms. This application is capable of displaying announcements and allowing NPS students to conduct daily musters.

THIS PAGE INTENTIONALLY LEFT BLANK

# TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

## LIST OF FIGURES

# LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

# LIST OF ACRONYMS AND ABBREVIATIONS

| | |
|---|---|
| ADT | Android Development Tools |
| API | Application Programming Interface |
| GUI | Graphical User Interface |
| HTML5 | HyperText Markup Language version 5 |
| IB | Interface Builder |
| IDE | Integrated Development Environment |
| MVC | Model View Controller |
| NIB | NeXT Interface Builder |
| NPS | Naval Postgraduate School |
| OpenGL | Open Graphics Library |
| OS | Operating System |
| OTA | Over the Air |
| SDK | Service Development Kit |
| UI | User Interface |
| WYSIWYG | What You See Is What You Get |
| XML | Extensible Markup Language |

THIS PAGE INTENTIONALLY LEFT BLANK

## ACKNOWLEDGMENTS

We would like to thank Dr. Thomas Otani and Mr. John H. Gibson, our advisor team. Thank you for your constant attention and motivation.

Corey would like to thank his lovely wife, Cathleen, without her support, encouragement, and understanding of the long nights spent writing, we would not have completed this thesis.

Christian would like to thank his parents for their support and motivation without it he would not have made it this far. He would also like to thank his friends for helping him remember to take a little time for himself every now and again.

THIS PAGE INTENTIONALLY LEFT BLANK

# I. INTRODUCTION

Mobile devices are fast becoming ubiquitous in today's society. The capabilities of these devices are increasing seemingly in accordance with Moore’s Law, becoming more powerful by the year. New devices are constantly being introduced to the market, each with a unique combination of computer architecture and software framework, or platform. As platform specifications are modified and improved upon, the gap between the platforms grows more pronounced.

## A. PLATFORM DIFFERENTIATION

There are a great number of mobile platforms currently on the market. In the purest sense each mobile device can be considered a separate platform. Each device, even if it is running the same Operating System (OS), is built upon different hardware. For the purposes of this thesis, we will classify platforms in a more general fashion, identifying them by the OS that they are running. This will limit the discussion of platforms to categories such as Android, iOS, and Windows Phone platforms. Additionally we will limit the scope of this thesis to discussions related to Android and iOS platforms.

## B. CROSS-PLATFORM DEVELOPMENT

Cross-Platform Development is the process of writing software applications for multiple computing platforms. Designing applications for multiple platforms is not a trivial task. There are several issues that must be overcome in order to release applications for multiple platforms. The most obvious difference between platforms is the language with which the applications are written. Additionally, developers should be aware of different hardware capabilities such as external SD cards and forward facing cameras. Finally, each platform has developed individual User Interface (UI) styles that users have become accustomed. Users expect that each application will adhere to the platform standard UI style. This final aspect all but mandates that cross-platform applications maintain separate UIs for each targeted platform.

In order to remain relevant in today’s application marketplaces, developers must embrace cross-platform development concepts to ensure that the applications are targeted to as many different platforms as possible. To that end it becomes necessary that any application developed for one platform also be made available for other existing with the ability to be ported to future platforms.

## C. PROBLEMS WITH CROSS-PLATFORM DEVELOPMENT

Applications targeted to iOS and Android platforms are written with completely different languages. Applications targeted for iOS are written in Objective-C while those targeted for Android devices are written in Java. The most obvious difference between platforms is the language with which the applications are written. Additionally, developers should be aware of different hardware capabilities such as external SD cards and forward facing cameras. Finally, each platform has developed individual User Interface (UI) styles that users have become accustomed. Users expect that each application will adhere to the platform standard UI style. This final aspect all but mandates that cross-platform applications maintain separate UIs for each targeted platform.

In order to remain relevant in today’s application marketplaces, developers must embrace cross-platform development concepts to ensure that the applications are targeted to as many different platforms as possible. To that end it becomes necessary that any application developed for one platform also be made available for other existing with the ability to be ported to future platforms.

Each platform consists of separate hardware profiles, including processor and memory, as well as screen size and other options such as cameras and Bluetooth. These hardware profiles cause the platform specific APIs to differ between platforms.

In addition to language and hardware differences, each platform provides unique user interface guidelines with which users have become accustomed and developers are expected to maintain in any application they develop

These factors result in increased costs in terms of time and money spent on the re-design process and the opportunity cost of that development time not being spent addressing new application design, or at the least, maintaining and upgrading the already released application. Additionally, the second design process often results in drastically different code bases that increases maintenance costs and may lead to applications with different features.

## D. CURRENT SOLUTIONS

An ideal method to accomplish this cross-platform dilemma is the ability to design and write a single application that runs on all platforms, or an interpretive cross-platform solution. OpenGL and Web Applications are both examples of this solution. These interpretive solutions can be run directly on each platform through the use of an interpreter. In the mobile realm the interpreters for Web Applications are the web browsers and for OpenGL it is the graphics libraries.

These solutions fall short of the desired cross-platform solution because they do not adhere to the platform specific UI Styles. A single UI is developed which may either correspond only to a single platform, or implement a platform neutral UI. Users from one or more platforms will be forced to adhere to a UI and navigation style that is unfamiliar to them.

Products such as Appcelerator’s Titanium and Corona provide third party APIs that result in separate but related applications tailored to specific platforms. While they have shown extremely promising results, we decided to pursue a solution that relied only on native platform APIs. We believe that developers are better able to handle security issues related to their applications by using native APIs and implementing applications in native platform languages without the aid of third party tools.

While these methods are accepted, and potentially cost efficient solution to the problem, we find that it lacks the customization that users of different platforms have come to expect from applications running on their devices. OpenGL and Web Applications force developers to choose a single UI and navigation schema that will be presented to users of any platform on which the application is hosted.

These neutral UIs and navigation schemas can often lead to user confusion on one or more of the platforms, as they do not provide common “look-and-feel” features with which users have become accustomed. Cross-platform applications should utilize interfaces specific to the platforms they are targeting to avoid such user confusion.

Short of relying on such “neutral” navigation solutions, we need to understand key similarities and differences of the platforms we plan to use. By identifying key similarities between the platforms we will be able to leverage those similarities to develop a design process that will use the aspects common between platforms while minimizing the differences, allowing for applications to be built for multiple platforms from a common set of design documents.

## E. UNIFIED DESIGN PROCESS

In this thesis we developed a Unified Design Process that simplifies the design process for cross-platform applications. We first discuss the concept of Design Patterns and how they may be used to solve commonly occurring design problems. We then discuss common approaches to mobile development, including common aspects of mobile application development, including navigation concepts and identification of individual screens that will be presented to users. Using these concepts, we discuss how to apply a set of patterns to different platforms. Finally, building on that knowledge, we lay out a design process, the Unified Design Process, which can be used to create applications that may be implemented on multiple platforms. We use this process to design and build the NPS Muster application, an application that could be used for NPS students to read announcements and conduct daily musters.

# II. BACKGROUND

This section provides an introduction into the Android and iPhone platforms, integral differences, and the issues related to the topic of cross-platform development of the two platforms. Additional considerations specific to platform architectures, Integrated Development Environments (IDEs), and design patterns will also be addressed.

## A. CROSS-PLATFORM DEVELOPMENT

Cross-Platform Development is the process of writing software applications for multiple computing platforms. These platforms can be described as a combination of computer architecture and software framework required to run software applications. An example of a platform would be the Microsoft Windows 7 Operating System (OS) running on the x86 architecture. The scope of this thesis will be limited to the Android OS and iOS mobile platforms.

Cross-Platform Development has been a topic of interest in Computer Science since the discipline’s inception. With the release of Apple’s iOS and, subsequently, Android OS, there has been an increase in the number of developers releasing applications tailored for specific platforms. It has become the expected norm for applications to be available on multiple platforms, but the differences inherent in the mobile platforms force developers to redesign their applications to work on different platforms. This can result in a significant increase of costs due to time and duplication of effort.

There are two ways of achieving a Cross-Platform Development solution. First is a developmental solution, which requires developers to design, build, and compile applications for each platform separately. The second is an interpretive solution, which can be run directly on each platform through the use of an interpreter. In the mobile realm an interpretive solution would be OpenGL or WebApps. Each platform has native software that will allow these types of applications to run. This thesis will focus on how to solve the mobile cross-platform problem developmentally.

## B. ANDROID PLATFORM

Android OS version 1.0 was release in September 2008, and has been quickly developed and incrementally updated. Version 3.0 was released in early 2011 and version 4.0 in late 2011.

Being open source, the OS has experienced significant branching in order to support many distinct platforms [1]. Figure 1 shows the branching of the Android operating system as of January 3, 2011. As the OS has evolved, to incorporate new platforms, certain API functions have become obsolete. Developers must realize which functions are not supported by older platforms and must tailor their applications accordingly. This gives the developers two choices; first, developers can choose to disregard certain versions of the OS. The second option is to re-write their code for each major release of the OS. Each application must be designed to run on all the various OS versions and be able to gracefully handle different hardware profiles, including screen size, screen density, memory size and processor speeds.

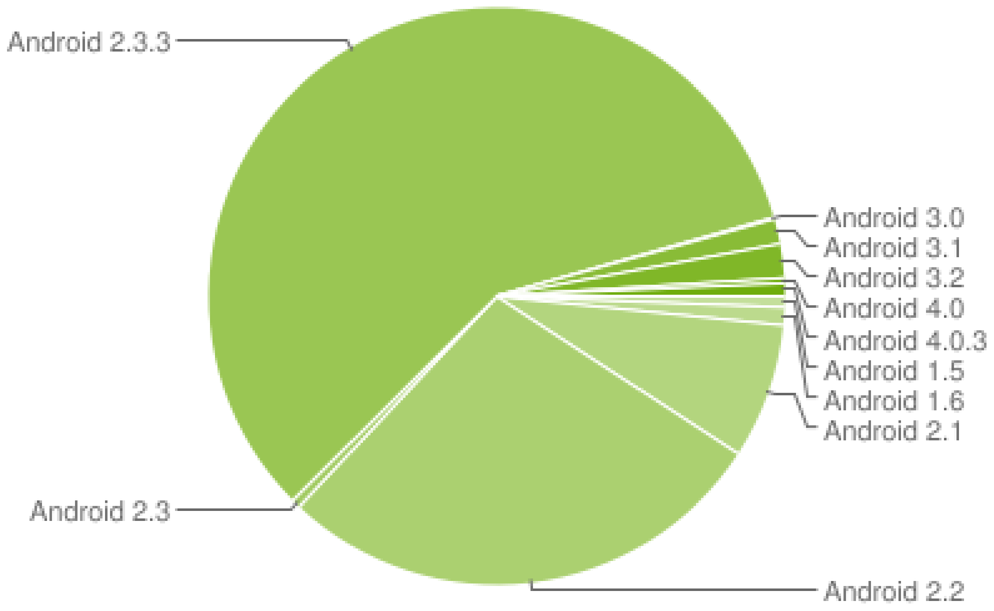


Figure 1. Android OS Branching (From [1])

### 1. Architecture

Android Architecture is based on the Linux 2.6 Kernel (Red Section of Figure 2). It is used as the hardware abstraction layer. The Linux Kernel provides a driver model, memory management, process management and other robust features that have been proven over time [2], [3].

The Android libraries (Green Section of Figure 2), written in C and C++, provide much of the core functionality and power of the Android platform. For example, SQLite is used as the core for the majority of the data storage needs. Webkit is an open-source browser engine and is the same engine that powers Apple’s Safari.

The main component of the Android Runtime (Yellow Section of Figure 2) is the Dalvik virtual machine. The Android platform was designed to meet the needs of an embedded environment, where battery, memory and CPU limitations exist. The Dalvik VM converts .jar and .class files into .dex files at run-time for a much more efficient byte-code. Additionally, .dex files are CPU optimized and designed to be shared across processes, resulting in the ability for multiple, concurrent Dalvik machines to be running on a single device. The Core Libraries (Blue Area of Figure 2) provide all the utilities that basic programmers require, such as File I/O, UI and basic functionality.

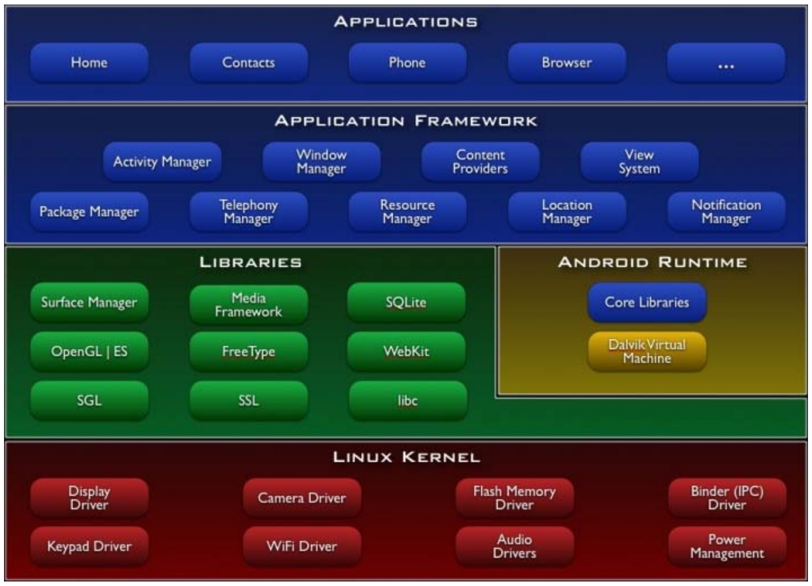


Figure 2. Android Architecture (From [2])

The Application Framework (Lower Blue Section of Figure 2) is the toolkit that all applications, Google or third-party developers, use. An example would be the activity manager. This application manages the application life cycle and a common back stack, enabling applications running in separate processes to have a smoothly integrated navigation experience. Content Providers are a unique piece of the Android Platform; they allow application to share data. An example of this is the sharing of contacts to any application with correct permissions.

The Application Layer (Top Section of Figure 2) is the actual implementation of the applications, such as the phone, messaging or our NPS Portal.

#### 2. Language

Android applications are written using a subset of the Java 6 SE API, replacing Swing, AWT and Applet classes with custom graphics and mobile development libraries. Google chose not to use Java VMs, and instead developed Dalvik, which allows each application to run in its own VM on the device.

#### 3. Development

Android can be developed on any platform (Windows, Mac, Linux) using any IDE that supports Java. The Eclipse IDE, however, has been optimized to support Android development. The optional Android SDK plugin can be added to the IDE providing increased functionality including support for a variety of Android Virtual Devices (Virtual Machines simulating different types of devices), What You See is What You Get (WYSIWYG) GUI editor, and the ability to easily sign releasable .apk packages from inside the IDE.

### C. IPHONE PLATFORM

iOS, formerly known as iPhone OS, was originally released in June of 2007.  iOS is based on Apple’s successful desktop OS, Mac OS X. OS X was cut down and modified to fit and run on devices with limited resources such as mobile phones. As the iPhone gained popularity, Apple began to release the OS on other mobile devices, such as the iPod Touch, Apple TV and iPad. With multiple device types running the iPhone OS, the OS was renamed “iOS.”

Apple is continually developing iOS to create a faster and smoother OS; the current version being 5.0.1. This version incorporates over 200 improvements, most related to the user experience. However, two major changes, unrelated to the everyday user experience, are the PC Free and the Delta update features.

iPods and all previous releases of the iPhone have always required the user to connect the device to his or her iTunes library, allowing the user to update software and manage content on each device. However, in iOS5, the PC Free feature removes this constraint. Users will now be able to set up and update the device without the use of a computer.

The Delta update feature allows the device to download application updates faster by requiring only the application changes to be downloaded. In previous additions, in order to update applications the full application needed to be downloaded and installed; this also applied to OS updates. With the upcoming OS the device will be able to download only the changes of the application or OS and install those on top of the previously installed application. This will reduce the amount of downloaded data per update. The decrease in the size of required downloads also paved the way for the new Over the Air (OTA) updates.

## 1. Architecture

iOS architecture is similar to that of the Mac OS X; at the highest level it acts as an intermediary between the hardware and on-screen applications. Applications do not directly communicate with the device hardware. Instead the applications communicate through a set of pre-defined system interfaces, allowing developer applications to work seamlessly on different hardware configurations. Although each application is protected against the different hardware configurations, developers still need to account for the configurations in their code. For example, a developer creating a photography application must account for the lack of a camera in the 1st generation iPod touch. This application will be able to use all the functionality of the app, except for taking pictures.

Within the iPhone OS (Blue Section of Figure 3), there are four major layers: Cocoa Touch, Media, Core Services and Core OS. The Core OS is built on the system level encompassing the kernel environment, drivers and low-level UNIX interfaces of the OS. The kernel, based on Mach, manages virtual memory, threads, file system, networking, and inter-process communications. Also, composing the Core OS layer is the Security Framework, External Accessory Framework and Accelerate Framework. These frameworks provide the necessary security, access to external

hardware, math functionality (basic math, big-number, and DSP calculations), and are optimized for iOS device hardware configurations.

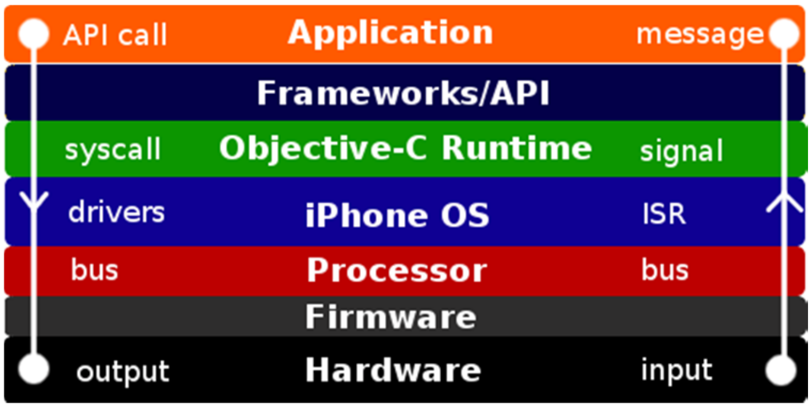


Figure 3. iPhone Architecture

The Core Services layer contains all fundamental system services; many parts of the system are built on top of this layer. This layer also includes support for SQLite databases, xml, grand central dispatch, and in-app purchases. Core service frameworks, such as the Address Book Framework, CF Network Framework (which manages Wi-Fi, cell, and Bluetooth networking), and Core Data Framework (model view controller data management) are contained in this layer.

The Media layer contains support for all audio, video and graphics technologies. This layer supports 2D and 3D graphics through the Quartz graphics engine. Various codecs, capable of decoding audio, video and AirPlay support, are also included in the media Layer.

Finally, the Cocoa Touch layer contains the key frameworks for building iOS applications. This layer defines basic application infrastructure and supports key technologies, such as touch-based input, multi-tasking and other high-level system services. This layer contains the implementation of the Apple view controllers. Developers will use support and services from this layer to create application UIs. Since this layer provides all functionality required by the typical developer, we will initially focus on this layer to begin identifying common programming aspects of iOS and Android.

### 2. Language

iOS is based on Mac OSX and utilizes the Cocoa Programming Environment. Cocoa automates many of the UI elements of an application so that applications developed in the environment conform to Apple’s human interface guidelines. The Objecive-C programming language is the basis for all iOS application development. Xcode offers developers all the editing and testing capabilities, as well as SDK documentation, in one quick and simple interface.

### 3. Development

Apple, in keeping with its proprietary nature, has limited iOS development to only Mac-based computers. The only authorized development IDE is Xcode. Currently, Xcode 4.2.1 is available for individuals registered as iOS developers who wish to develop in Cocoa, C++ and Objective-C. However, if a person enrolls in a developer program through Apple, Xcode 4.3 is available for download. The iOS Simulator is an expansive tool, which runs quickly and has nearly all functionality available by iOS devices. Apple’s Xcode, with the iOS API and SDK, provides a simple GUI for developing applications.

Apple has taken many precautions to ensure that its devices are running only approved applications from the App Store. Apple offers different developer programs to accommodate organizations and individual developers. If a large organization wants to develop in-house applications it must register for the iOS Developer Enterprise Program. The Enterprise program will offer the ability to release the in-house applications without going through the Apple iTunes Store approval process. The organization’s developers must also complete the individual developer process discussed below.

For an individual developer to install applications on a test device, developer profile must be created and registered. Also, the device must be registered with Apple. Once this is complete, the developer must then install the provisioning profile on the device. The device and provisioning profile then have to be added to Xcode preferences. Once this process is complete the device will work seamlessly with Xcode.

## D. DESIGN PATTERNS

In 1977, an architect named Christopher Alexander described a new concept in architecture in which he identified reoccurring problems and proposed solutions to those problems. He thought of these problem–solution pairs as a new form of architectural language that would allow even non-architects to begin designing their own houses, streets, and communities. He called this new language a “pattern language”; it uses words, drawings, photographs, and charts to describe patterns that can be used for designing anything from the placement of a doorknob to the construction of a skyscraper. Christopher Alexander says:

> Each pattern describes a problem which occurs over and over again in our environment, and then

> describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice. [4]

“The Gang of Four” applied the concept of patterns to object-oriented design in their book *Design Patterns: Elements of Reusable Object-Oriented Software*, where they defined the concept simply as “a general reusable solution to a commonly occurring problem within a given context” [4]. Design patterns are not a specific solution to a specific problem; they provide guidelines on how to solve general categories of problems. A description of a pattern should consist of at least the following items: pattern name, problem, solution, and consequences. The pattern name should be descriptive of the pattern. The problem should highlight the issue that the pattern addresses. The solution should examine a general method of dealing with the problem without going too deeply into detail. The consequences should identify costs and benefits associated with implementing the pattern.

A different architecture, different programming languages, and different design methods support each mobile platform. While work is being done to develop a common language and methods to compile code developed for one platform to be used on other platforms, it is imperfect and can lead to loss of platform specific capabilities and departure from platform specified user interface norms.

Patterns allow developers to conceptualize their designs at a higher level, focusing on non-platform specific implementations. They are able to identify which portions of their code can be easily ported between platforms, and which portions need to be tailored to suite specific platforms. Additionally, patterns will allow developers to decouple those portions that have been identified as platform specific from those that can be reused between platforms, increasing code reuse and allowing developers to take advantage of platform specific features; all while minimizing modification to the design of their core application designs.

### 1. Model-View-Controller Design Pattern

The Model-View-Controller (MVC) pattern (Figure 4) is actually a compound pattern composed of several separate patterns, each addressing a unique problem. Variations of this pattern have been used to build user interfaces since Smalltalk [5]. It consists of three classes: a Model, which represents application logic; the View, which is the screen representation of the Model; and the Controller; which pre-defines how the View will react to user input [4].

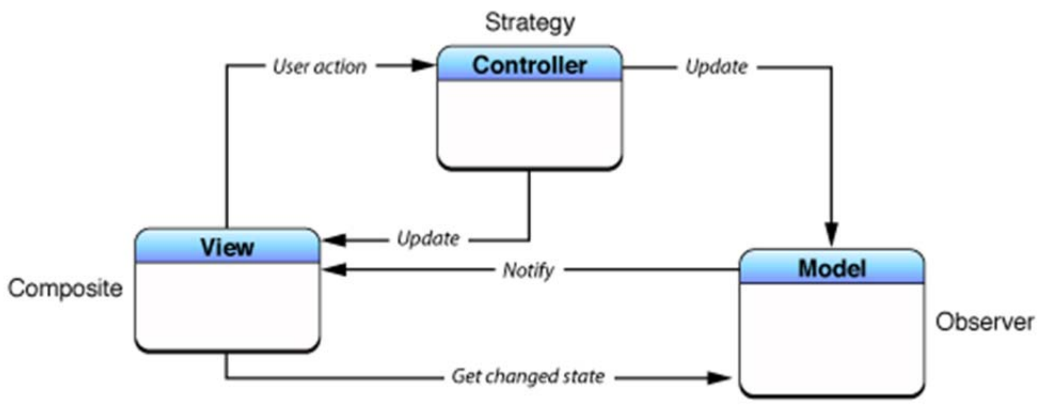


Figure 4. MVC Pattern (From [4])

The View uses the Composite pattern that allows individual View objects, such as buttons and text boxes, to be grouped into tree structures, which can in turn be treated as a single View in the same way that any individual View would be addressed. If you call the draw() method on the composite structure, it will in turn call the draw() method on all of its children, regardless of whether they are leaf elements or nested composite Views. This allows client code to treat composite elements the same way they would treat primitive objects, they need to know very little about the View elements with which they are interacting.

The Controller implements a Strategy pattern in which a family of algorithms is defined, encapsulated, and made interchangeable. The Controller contains the logic for how the View reacts to user actions. Take for example, a media player with a single button on the screen. In the “stop” state the button would be expected to start the .mp3 (Model), and that is what the Controller does when it responds to the button click. Once the .mp3 is started, there is no reason for it to start the media player again, so the controller can be replaced with one that has an identical interface but instead of causing the model to start, it causes it to stop, at which point the original controller can be utilized again. The Strategy pattern allows algorithms to vary independently of clients that use them; it allows views and controllers to be decoupled and modified independently.

The Model is an example of the Observer pattern, which is a one-to-many relationship between objects in which many objects can subscribe to a single Model in order to be notified when it changes states. Instead of constantly polling Models to ensure that they have the most recent representation, Controllers can subscribe and be notified by the Model whenever it changes state. The “Gang-of-Four” envisioned a MVC pattern in which the View also subscribed to the Model and was updated independently of any associated Controllers, as presented in Figure 4. Implementing the Observer pattern frees system resources, sending messages between objects only when model data has been changed. If over-used, there are cases where Models may update their subscribers when unrelated data has changed.

#### 2. MVC and Mobile Applications

iOS relies heavily on the concept of MVC. It has re-envisioned the pattern in such a way that the Model and Views are unaware of each other. Android does not enforce the use of this pattern, but we have developed a design process that will allow us to view Android applications in the light of the MVC pattern.

## E. CONCLUSION

At first glance, it seems as if the iOS and Android platforms are significantly different. They do not share a common language nor do they share a common architecture. It is difficult to envision a process in which developers may utilize one set of application design documents during development. Chapter III will discuss the basic UI building blocks of mobile applications as well as generic navigation concepts. This will identify several common concepts useful to the development of a Unified Design Process. Chapters IV and V will show how a common MVC pattern may be applied to iOS and Android platforms. Chapter VI will utilize these concepts and techniques to describe a process that will allow developers to design an application for any mobile application. This Unified Design Process was tested during the development of an NPS Muster application.

THIS PAGE INTENTIONALLY LEFT BLANK

# III. NAVIGATION AND USER INTERFACE COMPONENTS

On the surface, the Android Operating System and iOS user experiences are markedly different. Each provides a distinct user experience that is expected to carry through to third-party applications in order to maintain a consistent user experience. The underlying structures on which the individual UIs are built are remarkably similar. By understanding the similarities of these basic UI and navigation components, we are able to apply a level of abstraction that allows developers to build applications, targeted for multiple platforms, from a common design document. This chapter will identify the common concepts of navigation and User Interface (UI) building blocks shared by both operating systems.

## A. UI BUILDING BLOCKS

Mobile applications are, by nature, UI intensive. Each platform has a distinct style that users expect developers to incorporate into their designs. When users buy a specific phone they also buy into the platform UI style. If applications differ from the platform specific UI style users can become confused or frustrated. Thus, it is good practice for developers to design their application UIs targeted for each platform.

Each platform has a distinctive style that developers should follow in their applications; the core elements used to design the UI for any platform are similar. If one understands the similarities and differences of the basic UI building blocks, they should have no trouble creating multiple UIs that fit with the targeted platforms, but maintain similar functionality.

### 1. View

Android and iOS both utilize the concept of a View as the base of every user interface. Android’s View [6] and iOS’ UIView [7] classes both define rectangular areas on the screen. This interface manages the displayed content and event-handling code necessary to process user interactions. Figure 5 is a visual representation of a view hierarchy in Android (left) and iOS (right). Views rely on the Composite pattern as discussed in Chapter II. Each view object may contain zero or more subviews each responsible for positioning and sizing inside of itself. All building blocks discussed in this section are subclasses of View or UIView classes.

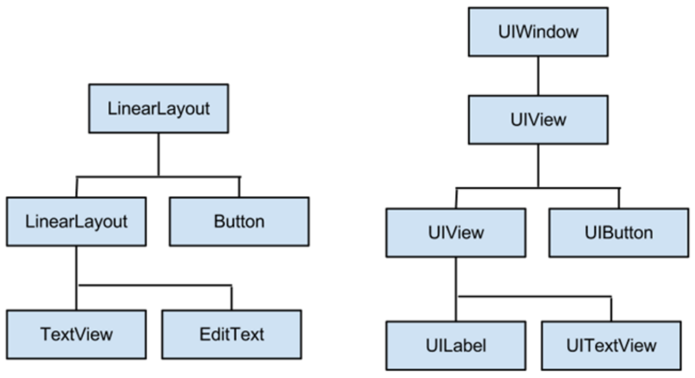


Figure 5. Example View Structures

### 2. Buttons

Buttons provide intuitive connection to the application functions by enabling a user to interact with the UI through a simple press action. Buttons are realized in Android as an instance of the Button class and in iOS as an instance of the UIButton class. Figure 6 is an example of the basic button used by Android (left) and iOS (right).

The main difference in implementation between iOS and Andriod is that an instance of UIButton intercepts touch events and actively calls a function in a target object, while Android objects establish a listener that takes action when they receives a callback event from a Button. We have found that during design the difference is negated by identifying the function which would normally be called by the UIButton in iOS and placing a call to that function inside the OnClickListener associated with the Button in Android.

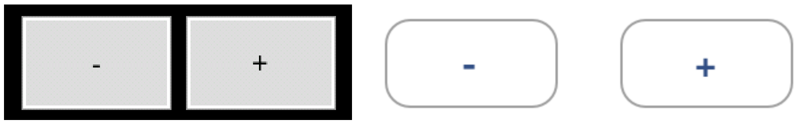


Figure 6. Android and iOS Buttons

### 3. Text Fields

Text fields are integral to any application that either displays static data or receives text input from a user. Both platforms provide several specialized subclasses to edit text, but the simplest of these classes are UITextView in iOS and TextView in Android. Each of these Views provide added functionality, such as a user being able to modify text content through the use of an on screen keyboard, and character masking for when a user is entering a password.

### 4. Spinners

Android’s Spinner and iOS’s UIPickerView are Views that provide an intuitive method for displaying the currently selected item from a list of similar items, as well as simple methods for users to choose that selection. Both classes rely on an Adapter (Android) or Delegate (iOS) to handle the underlying data and the individual views associated with each row in the Picker UI. Figure 7 shows the basic spinner elements for Android (left) and iOS (right).

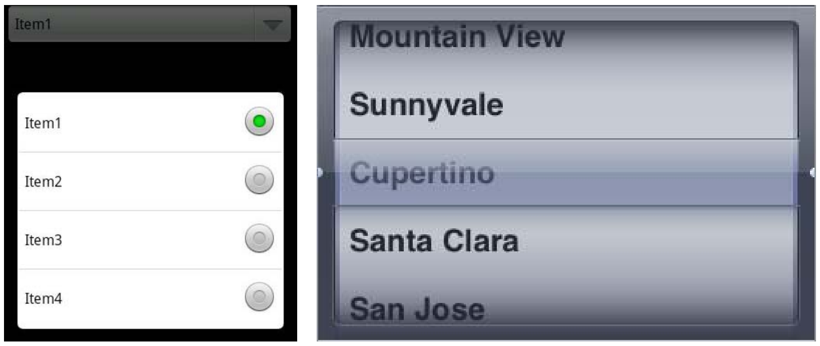


Figure 7. Android and iOS Spinners

### 5. Lists

Information is commonly organized and displayed in lists. There are several methods available on both Android and iOS that allow an application to display these lists in meaningful ways to the user. Lists (Figure 8) can be used for data selection and drill-down (Stacked) navigation.

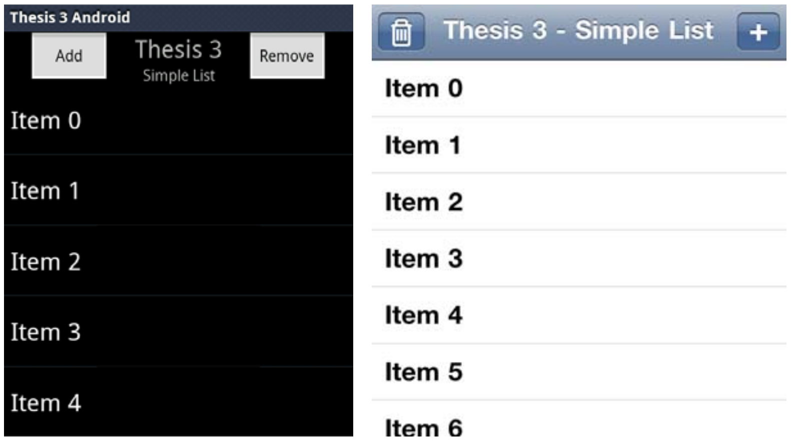


Figure 8. Android and iOS Lists

### a. *Selectable List View*

Android utilizes the ListView class and iOS uses the UITableView class to provide basic functionality for displaying multiple line items in vertical arrangements. Both classes require a data source and an Adapter (Android) or Delegate (iOS) to format the individual item views presented in the list. Both platforms provide the ability to set custom views as list items; these custom views can be made up of other View elements such as Buttons, Images and Text Views.

iOS provides detailed guidelines regarding implementing list-items. The UITableViewController class is used to manage a UITableView and its associated data source, which is typically a NSMutableArray. The UITableViewController method tableView:cellForRowAtIndexPath will define the allocation and setup of individual table rows. The tableView:didSelectRowAtIndexPath method will define functionality associated with row selection.

Android provides differentiation of short click for selecting an item and long press for accessing other options for an item such as editing or deleting from the list view.

#### *b. Multi-Select List View*

Both platforms provide list views that incorporate check boxes, expanding the functionality of Selectable List Views in order to allow users to choose several options from a list at one time.

### 6. Switches

Switches, shown in Figure 9, are a subclass of button on both platforms. They may exist in one of two states, on or off, returning Boolean values. iOS uses the UISwitch class where Android uses the Switch class.

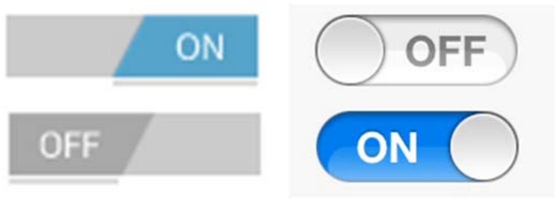
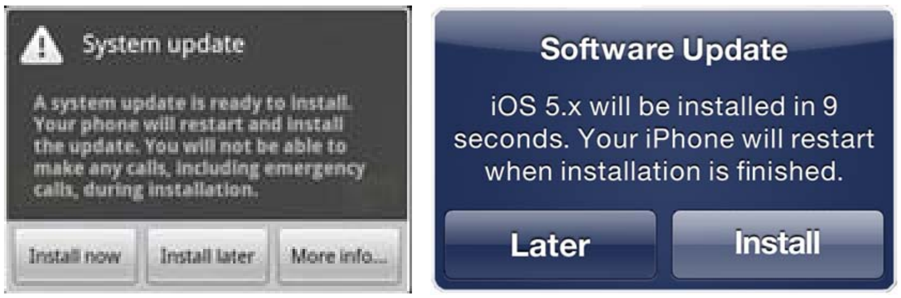


Figure 9. Android and iOS Switches

### 7. Dialogs

Dialogs are small Views that appear in front of Current Views to display an alert message to the user, such as the System Update message in Figure 10. iOS implements the UIAlertView class and Android uses the Dialog class. Both classes are able to modify the title, message and Buttons presented on the dialog. The Dialogs can also be customized to show information other than messages.


System update

A system update is ready to install. Your phone will restart and install the update. You will not be able to make any calls, including emergency calls, during installation.

Software Update

iOS 5.x will be installed in 9 seconds. Your iPhone will restart when installation is finished.

Figure 10. Android and iOS Dialogs

### 8. Radio Buttons

Radio buttons are two state buttons that can either be checked or unchecked. Radio buttons in an unchecked state may be clicked and become checked, but further clicks do not uncheck the radio button. Radio buttons are often grouped together to extend functionality. Clicking one Radio button in a group automatically deselects other grouped Radio Buttons. Android implements the RadioButton class while iOS implements the UISegmentedControl class. While a Segmented Control does not resemble a standard Radio Button, the underlying functionality is roughly the same. Figure 11 shows the Android Radio Buttons on the left and the iOS Segmented Controller on the right.

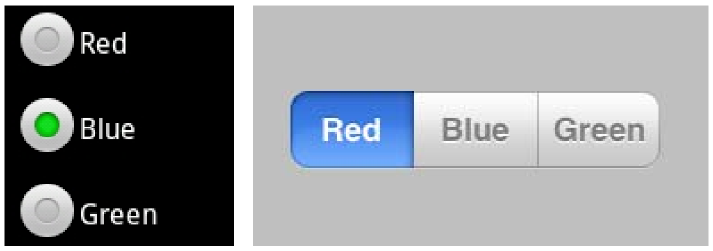


Figure 11. Android and iOS Radio Button

### 9. Progress, Sliders, and Loading Activity

Progress Views (Figure 12) show the progress of an operation. There are also activity indicators available for both platforms that indicate to users that a task or process is progressing without actually indicating percentage complete. Android implements Progress Bars through the ProgressBar class and iOS uses the UIProgressView class.

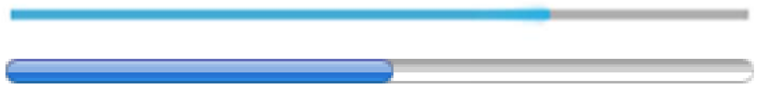


Figure 12. Android and iOS Progress Bars

Slider Bars allow a user to select a single value from a continuous range of values. In Android the Slider Bar is implemented using a SeekBar, which is an extension of a ProgressBar (Top Bar of Figure 13). In iOS it is implemented using a UISlider (Bottom Bar of Figure 13).

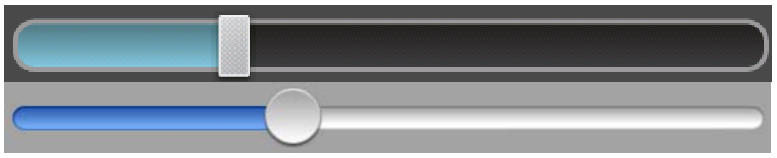


Figure 13. Android and iOS Slider Bars

Loading Activities are animations that are displayed in both Android and iOS to show that the device is processing user input. Often times the Loading Activity is used when sending or receiving information from a server. Figure 14 shows the Android and iOS Loading Activity.

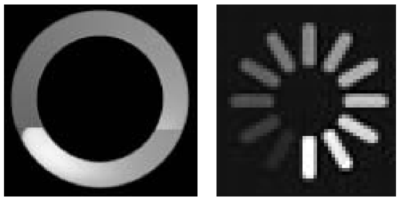


Figure 14. Android and iOS Activity

## B. NAVIGATION

Mobile devices are typically much smaller than their desktop counterparts and thus have significantly less screen real estate. Where typical computer programs may be able to show all information on a single large screen, mobile applications are presented to users in a modular fashion. The visual aspects of these modules are called views, while the underlying code components for Android and iOS are called Activity and View-Controller, respectively. For the purpose of this paper, the combined code and view components will be referred to as “screens” and the underlying code will be referred to as “controllers.”

Most applications will utilize multiple screens and the concept of switching between those screens is considered navigation. Application navigation can fall into one of several groups: Null, Stack, Horizontal navigation, and a hybrid that combines aspects of these three fundamental techniques.

### 1. Null Navigation

In both Android and iOS, it is possible to have multiple views associated with one controller. The controller is responsible for switching the different views to the forefront, presenting underlying data, and event handling for each view. The details of the presentation and event handling interaction between the controllers and views will be discussed later in this chapter.

Another integral part of application design is the presentation of screens and the interaction between screens and controllers. These topics will be covered in Chapters IV and V. In Null Navigation (Figure 15), the one controller is associated with multiple views and will explicitly determine which one to display. In any non-trivial application this method becomes unwieldy. A single controller will contain all code necessary for presenting and switching all views associated with that application. This leads to significant problems with readability and does not lend itself to good object-oriented practices.

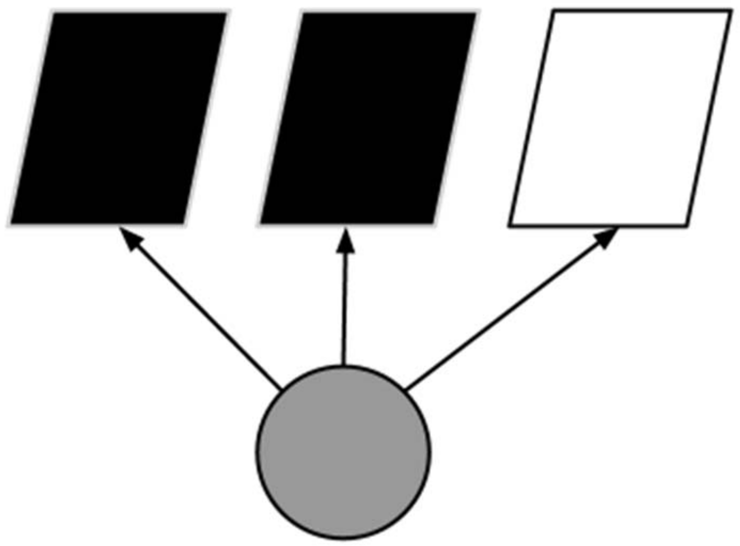


Figure 15. Null Navigation

Generally, Android and iOS both recommend modularizing application code in such a way that each view is paired with one code module [8], [9]. In more advanced applications, it is possible to combine these modules to create more intricate designs. For the purpose of this thesis, we will use a one-to-one relationship between code modules and view hierarchies.

### 2. Stack Navigation

One of the more modular approaches to navigation is the concept of “stacking” screens. In this method, an application would act as a controller implementing navigation by pushing screens on to a stack as a user navigates deeper into the application. This creates a hierarchical design that allows for reverse navigation, by popping the topmost screen from the stack.

Stack Navigation (Figure 16) is considered good practice because it logically separates screens into functional entities. (TASKS in Android) Applications consist of one or more such loosely related screens, consisting of one controller and its associated view. The application is responsible for communication between individual controllers.

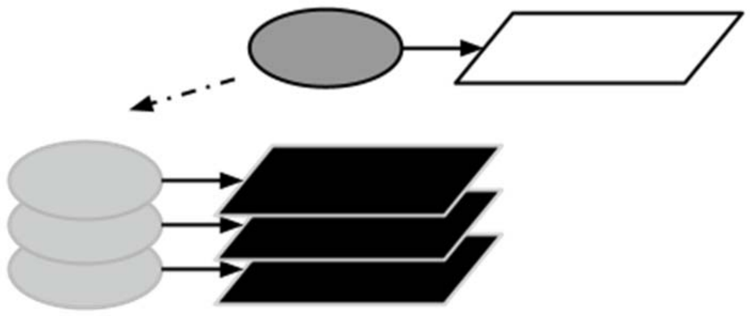


Figure 16. Stack Navigation

### 3. Horizontal Navigation

Horizontal Navigation (Figure 17) is realized through the use of tab bars. Each screen is displayed at the same level as the other screens without reference to display order. Different screens can be displayed at any time based on tab selection. Unlike stack navigation, there is no ability to backtrack, each screen can be thought of as active, constantly displaying its information and shown as necessary.

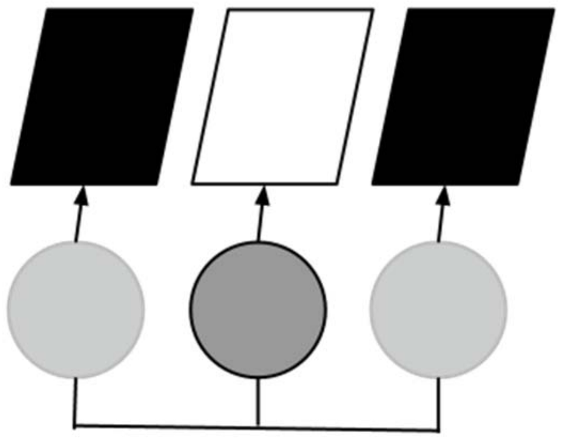


Figure 17. Horizontal Navigation

### 4. Composite Navigation

Composite Navigation (Figure 18) combines Horizontal and Stack Navigation by either stacking on top of tabs or by nesting a Horizontal Interface on top of a Stack. By combining the two types of Navigation, developers are able to create naturally flowing apps where users are capable of navigating into multiple view hierarchies.

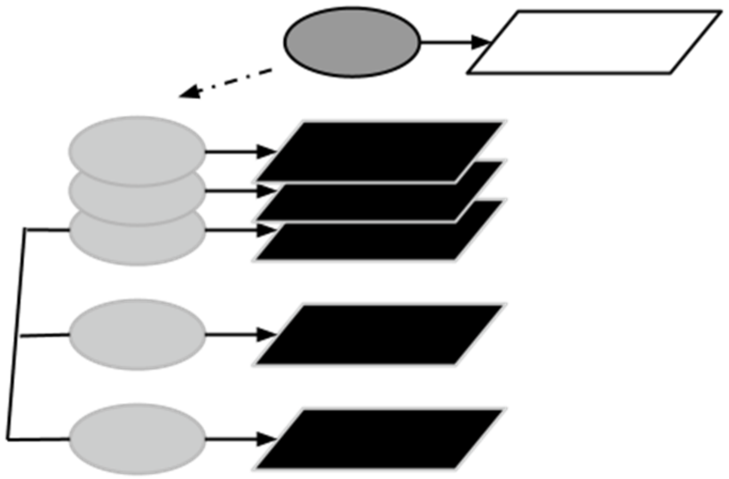


Figure 18. Composite Navigation

Typically, Composite Navigation has a Horizontal Navigation pattern at the root with Stack Navigation embedded into each node. A developer is able to place Horizontal Navigation within the stack, creating a branching effect, but this can lead to user confusion, especially when Tab Bars are placed at different levels of a stack. The adverse impact of user confusion may be offset by the value of the added functionality acquired from combining horizontal and stacked navigation. The apps we developed are typically built using Composite Navigation.

## C. CONCLUSION

Android and iOS platforms provide specialized UI experiences with which users have become accustomed. These experiences are expected to be used by third-party applications, which prevents developers from reusing View designs between platforms. Applying a layer of abstraction to the design process can largely marginalize these differences. Understanding there are few differences between the basic UI building blocks and Navigation principles between the two platforms allows application design to be broken into similar functioning classes.

Identifying these similarities at an abstract level will allow us to apply patterns when designing new applications for cross-platform deployment. The patterns we suggest will highlight these similarities, allowing the consolidation of application logic into specific classes that can be reproduced on either platform. These patterns will also highlight areas that may vary between the platforms. By understanding where variability occurs, developers will be able to minimize inconsistencies between the implementations of their applications.

Chapters IV and V will discuss the application of a common design pattern, the Model, View, Controller, in iOS and Android applications. These chapters will also demonstrate how that pattern can be used to design a single screen on either application. Finally, the concept of application lifecycles for the targeted platforms will be discussed.

# IV. IOS DEVELOPMENT

This section details the iOS development process and the implementation of the Model View Controller (MVC) Design Pattern in iOS applications. By default Xcode, Apple’s IDE, integrates the MVC pattern into every project. Developers are required to have an understanding of the components of the MVC in order to create applications. The following sections will break down each part of the MVC and introduce how iOS handles the application and ViewController lifecycles.

## A. INTEGRATED DEVELOPMENT ENVIRONMENT

The most recent version of Xcode binds the code development and Interface Builder (IB) processes more closely than prior versions. The code development portion provides the standard editing and documentation functionality of other professional IDEs. The IB provides a simple UI for developers to create application UIs by supporting the “drag-and-drop” of pre-defined UI elements onto base views. Furthermore, IB allows developers to link those UI elements to Controller classes through the integrated File Investigator.

Xcode provides several pre-defined project types through which one may begin building an application. These application types are: Master Detail, OpenGL Game, Single View, Tabbed, Page-Based, Utility, and Empty. For the purpose of this thesis, all projects were started as Empty Applications, the most basic of the options. When this type of project is created, a developer is only given the

AppDelegate.h and AppDelegate.m files. This option gives the developer more flexibility when creating the application UI.

iOS uses Objective-C; as with other C languages, classes are defined using implementation files, denoted by the .m extension, and header files, denoted by the .h extension. iOS begins all projects with an AppDelegate that controls the overall application lifecycle, discussed later in the chapter.

Screens are created by subclassing the UIViewController class and creating a NeXT Interface Builder (nib) file. Xcode will handle this process providing a minimal subclass and blank Views in associated nib files.

## B. NAVIGATION

Navigation Controllers in iOS align well with the general navigation concepts discussed in Chapter III. The AppDelegate instantiates the base window, which will contain all subviews, and the root navigation controller. The root navigation controller provides the functionality of screen switching within the application. The following sections highlight how iOS implements each of the general navigation concepts as a root navigation controller.

### 1. Stack Navigation

In iOS, Stack Navigation is realized by implementing a UINavigationController. A UINavigationController will provide basic navigation UI elements such as a Navigation Bar (see Figure 19). The Navigation Bar provides references to label the current screen as well as additional navigation options such as a “back” button to pop the current screen from the stack.

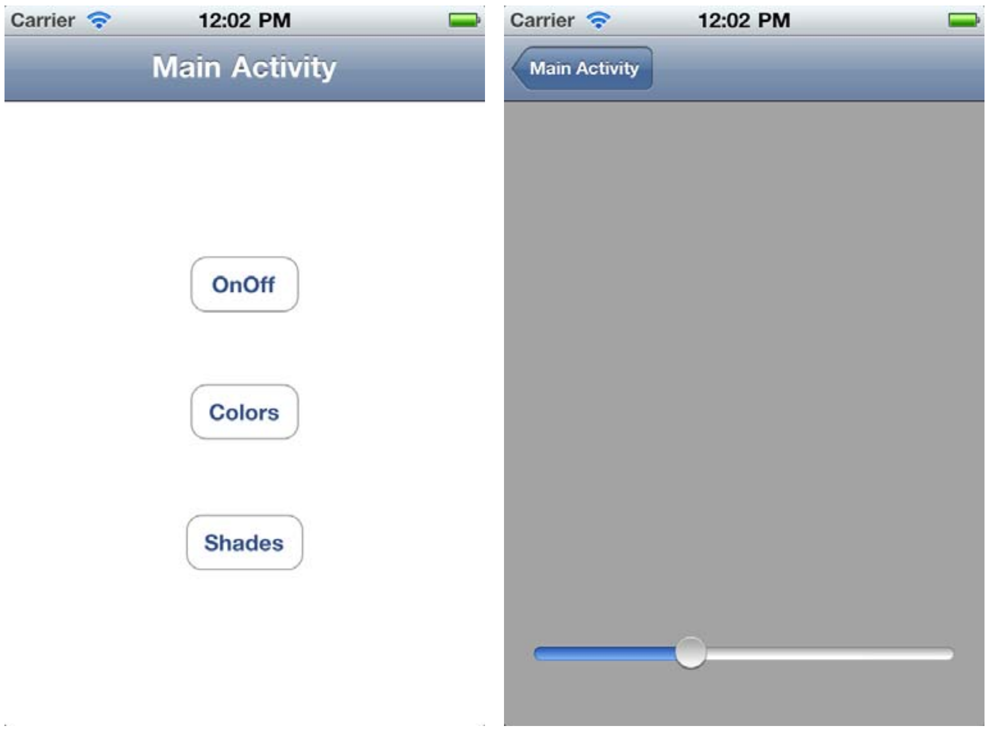


Figure 19. iOS Stack Navigation Screenshots

### 2. Horizontal Navigation

iOS implements Horizontal Navigation through the use of a UITabBarController. Developer documentation states that:

> The view hierarchy of a tab bar controller is self contained. It is composed of views that the tab bar controller manages directly and views that are managed by content view controllers you provide. [10]

This means that a UITabBarController manages not only a view, but also several Content Controllers, each of which may be a UINavigationController that could manage its own stack of UIViewControllers. The tab bar controller provides a view element, which consists of two or more tabs on the bottom of the screen and a blank view area where Content controllers can display custom views. The tab bar controller is responsible for managing the navigation between its child controllers, ensuring that the correct Content controller is presented when the user chooses the associated tab.

### 3. Composite Navigation

The most common form of Composite Navigation in iOS is realized by embedding UINavigationControllers within a UITabBarController. Figure 20 shows the transition into a deeper level of the hierarchy within a tab bar.

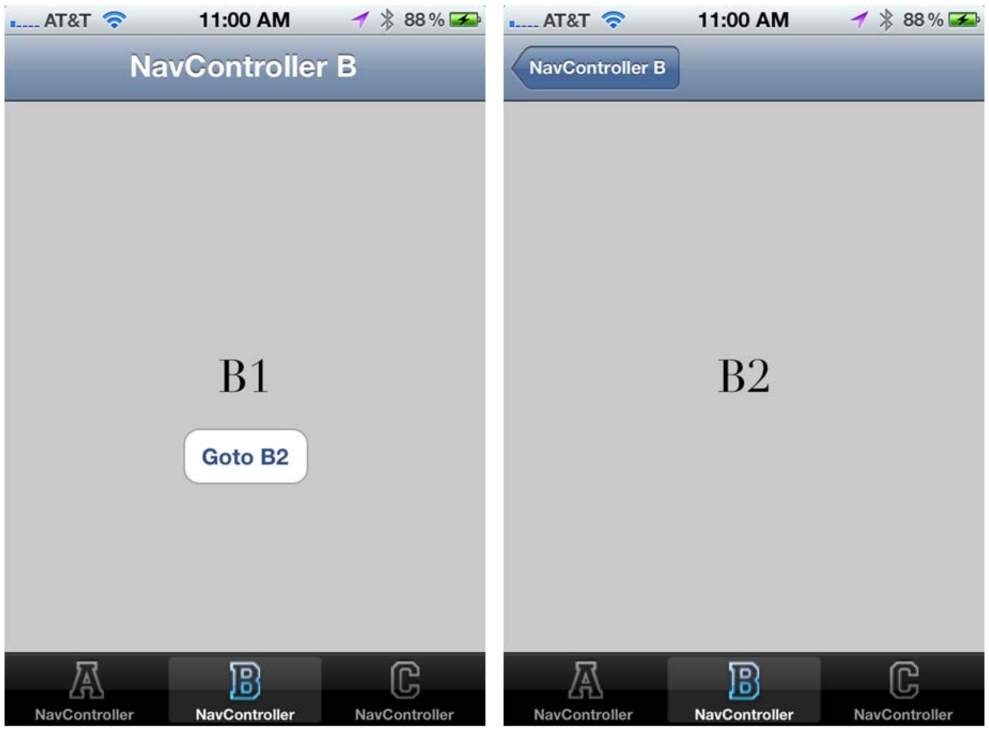


Figure 20. iOS Composite Navigation Screenshot

It is possible to embed a UITabBarController within a UINavigationController, but is not recommended by Apple. The Apple Developer Library states:

> A navigation controller can incorporate custom view controllers, and a tab bar controller can incorporate both navigation controllers and custom view controllers. However, a navigation controller should not incorporate a tab bar controller as part of its navigation interface. The resulting interface would be confusing to users because the tab bar would not be consistently visible. [11]

Generally, we agree with this statement; however, we have found that it may be necessary to employ such a design in an application. If done properly, and not overused, tab bars may be nested inside a Navigation Controller without creating user confusion.

## C. MODEL-VIEW-CONTROLLER DESIGN PATTERN

Apple developer documentation describes a re-envisioned MVC pattern. The benefits of adopting this pattern are re-usability, better-defined interfaces, and simpler extensibility [12]. iOS decouples the View from the Model, as shown in Figure 21, by using the Controller to handle user actions received by the View and to update the View as the Model changes. The View does not need to know the Model exists and vice-versa. The View and Model can be modified or replaced independently affecting only the Controller. In iOS, a Model can be any Object that represents or holds data of some variety, the Controller is a ViewController, and the View is a Composite View defined by an .xib file. A View can be linked to the Controller through the definition of IBOutlets in the header file and connected in the IB through the File Inspector.

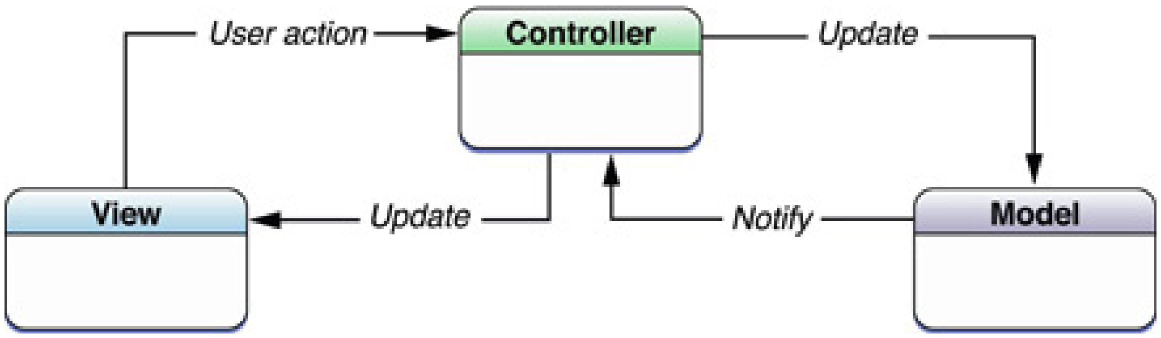


Figure 21. iOS Model, View, Controller Pattern (From [12])

In order to discuss the application of the MVC pattern, we present a simple application that simulates a beating heart. The Beating Heart app exemplifies the many-to-one relationships between models and Controllers, and the one-to-one relationship between Controllers and Views. The code for this application can be found in Appendix A. There is one Heart Object, two Controllers, and two Views. The App has two screens displayed inside of a tab bar (Figure 22). One screen displays an image of a beating heart while the other provides settings to change the frequency at which the heart beats.

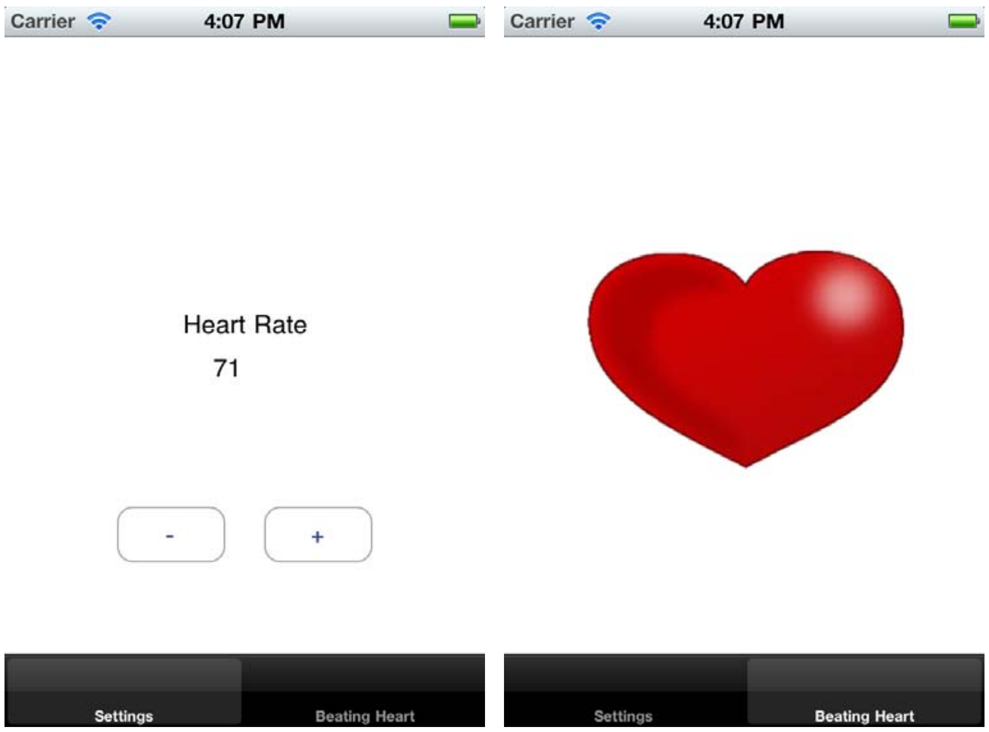


Figure 22. Heat Beat iOS Screen Shots

### 1. Model

“Model objects encapsulate the data specific to an application and define the logic and computation that manipulate and process that data” [12]. In our Beating Heart application, we created a class named Heart that contains all data and logic of a beating heart. This class contains no links to the UI and provides no functionality for communicating directly with a user.

The Heart class contains two integer data elements heartRate and beatCount, which determine the heart beat frequency and maintain a running count of the number of times the heart “beat.” Along with the integer elements, it contains the corresponding getter and setter methods that are publicly accessible. This model also contains the logic required for it to simulate a beating heart. A function called “beat” will increment beatCount and then set a timer for itself according to the heartRate value. After the timer has completed it will then call the “beat” function again.

In this project, the views implement an observer pattern. The details of the interaction between the views and the model are explained in the following section.

### 2. View

As stated earlier, Views follow the composite design pattern, implementing a hierarchy of Views and Sub-Views. Each application contains one root view in the hierarchy, called the window. The window serves to “manage and coordinate the [views] an application displays” [13].

A view is typically an instance of a subclass of UIView. It can be instantiated programmatically or through the use of nib files created in Interface Builder (IB). For most applications, the IB provides all necessary functions for building aesthetically appealing, intuitive, and functional UIs. It provides simple and intuitive drag-and-drop functionality for creating View object hierarchies and linking the View objects to View Controller methods. For more dynamic UI development iOS provides features for creating View Objects programmatically along with the ability to place them inside view hierarchies at run time.

Our Heart Beat application utilizes two View hierarchies, one that displays a UIImageView of a beating heart, and one that contains two UILabel objects and two UIButton objects.  One UILabel is simple text that remains static while the other displays the Beats Per Minute at which the Heart Object is currently set. The UIButtons send events to corresponding IBOutlets based on user interaction. The link between the IBOutlets in the code and the UIButtons on the screen are created through a drag-and-drop system in the IB. When the user clicks the window in the area of the button, the button will intercept the touch event and send the event to the associated method in its controller. The controller will then update the model to reflect the user input by either incrementing or decrementing the heart rate variable. After updating the model, the controller will then update the UILabel that displays the current heart rate.

View objects are ignorant as to the existence of any model objects and vice versa. The View’s only responsibility is to pass user touch events to its associated Controller and to display information as requested by that Controller.

### 3. Controller

Controllers tie the MVC pattern together, communicating between Model and View objects. Touch events processed by View objects, such as UIButtons, are associated to specific methods in the Controllers. When a user presses a button the corresponding method in the Controller is called and the Controller reacts accordingly.

Controllers may be associated to one or more Model objects. Depending on the situation, the association may be loose, wherein the Controller will poll the Model when it needs to access the underlying data or logic, or the association may be tight, implementing another Observer pattern. By registering as an Observer, the controller will be notified of any relevant changes.

We have implemented two Controllers in the Beating Heart application. Both controllers have a reference to a single instance of a Heart model. The first controller is the HeartView class. This class registers itself as an observer of the beatCount variable using the function registerAsObserver. Whenever the beatCount changes the observeValueForKeyPath function is called and the image of the heart is briefly changed to a larger version, animating a heart beat.

The second controller is the HeartSettings class. This controller contains the functionality for updating the heart rate of the model. It has two functions that increment or decrement the heartRate variable within the model. These functions are linked to the view buttons, mentioned above, through the IB. We also implemented the Observer pattern in this class. Every time the Model’s heartRate variable is modified the observeValueForKeyPath method is called and the Controller updates the UILabel with the current heart rate.

The Beating Heart application exemplifies the use of the Model, View, Controller pattern as well as the Composite Pattern and the Observer Pattern. We showed how one Model may be accessed by multiple Controllers and that each View has a single associated Controller. In the next section, we will discuss the concept of application lifecycle as it pertains to an iOS application and individual controllers.

## D. LIFECYCLES

Every iOS application has a designated starting point; it will run continuously until ended, either by the user or by the OS. Once it has begun execution, the application must account for the many activities that can happen on a mobile device. A majority of the applications will run to completion, but there may be events that interrupt that process, such as phone calls or text messages. In order to account for these situations, iOS has built-in methods that, when properly coded, will create smooth application transitions.

Within the application, each UIViewController has its own lifecycle. The switching of screens triggers different method calls. If improperly handled, these events can cause application faults. In the next sections, Application (Figures 23 and 24) and ViewController lifecycles will be explained.

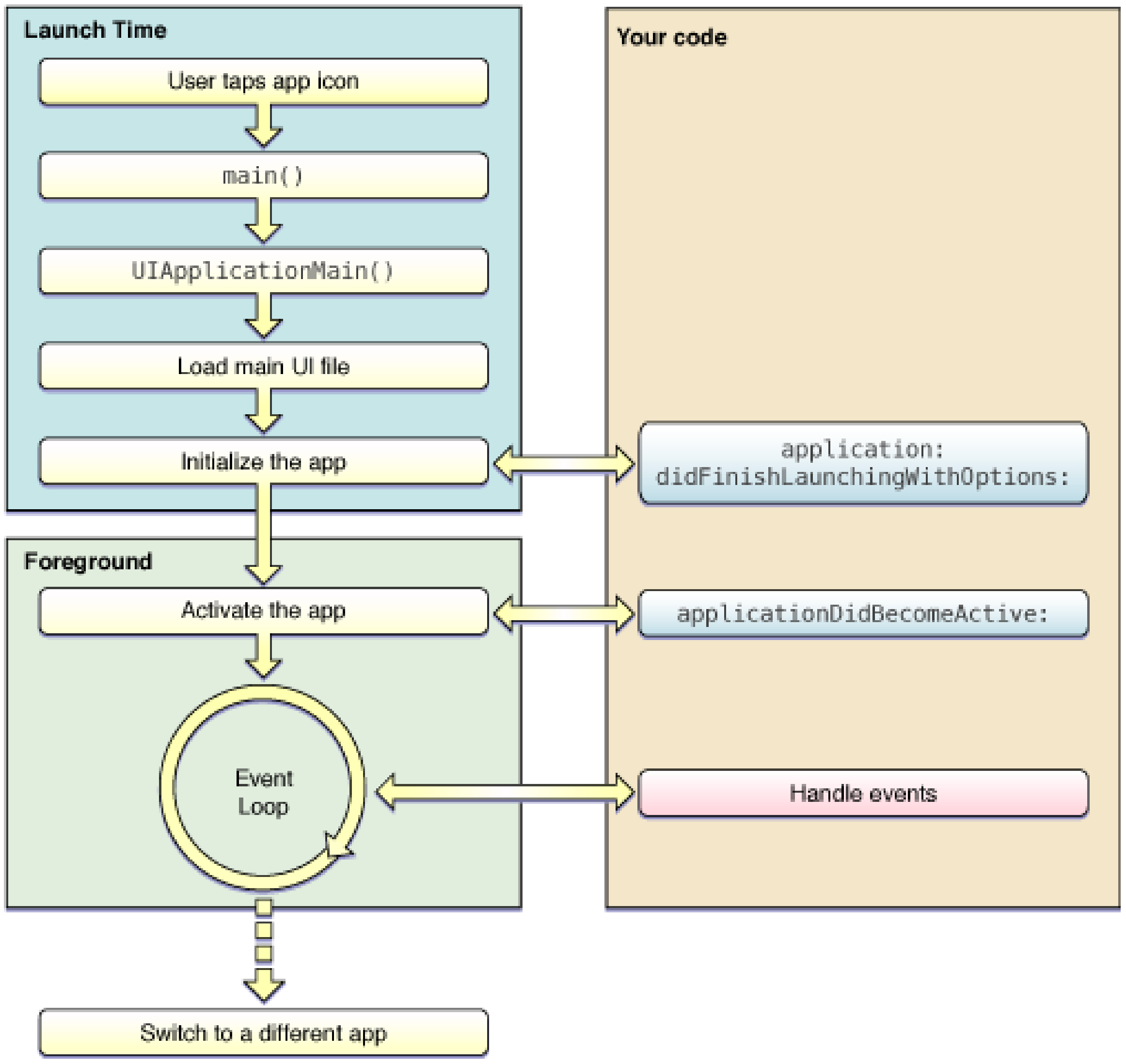


Figure 23.    Application Lifecycle (From [14])

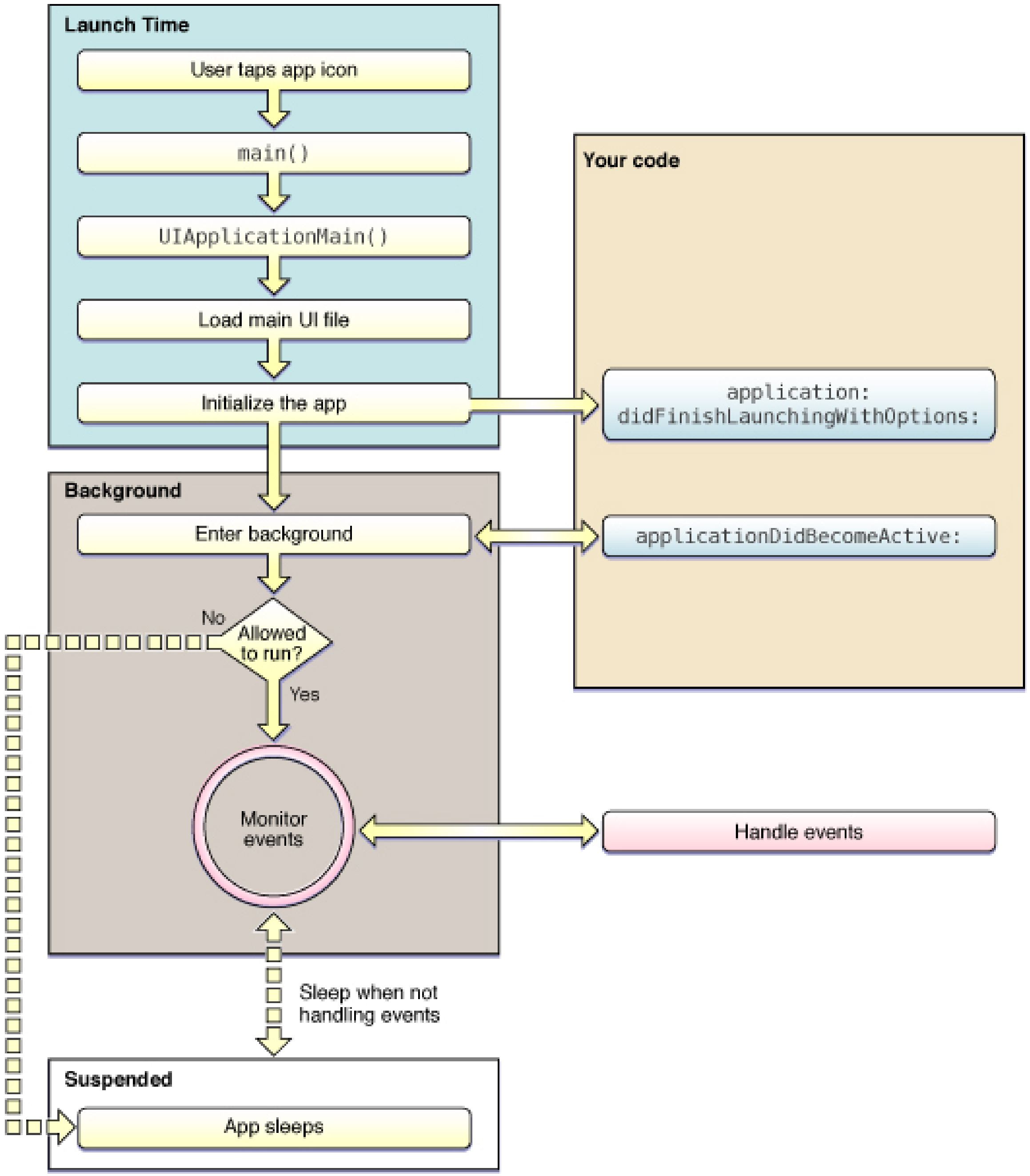


Figure 24. Application Launch into Background (From [14])

### 1. Application Lifecycle

Each iOS application has one instance of UIApplication. This is an example of the Singleton pattern in which this instance may be accessed subsequently from different controller objects by invoking the sharedApplication class method. “The UIApplication class provides a centralized point of control and coordination for applications running on iOS” [15]. A developer may access the UIApplication code in the main.m file that comes pre-configured with every new project. The application object is assigned a UIApplicationDelegate that is informed of significant run-time events. The application can be in one of five states: Not Running, Inactive, Active, Background, Suspended. Apple documentation defines these states as [14]:

- Not Running—The application has not been launched or has been terminated by the OS.
- Inactive—The application is running in the foreground but is not receiving events.
- Active—The application is running in the foreground and receiving events
- Background—The application is executing code in the background
- Suspended—An application is in the background not executing any code

iOS provides the developer methods which allow customized functionality for transitions between states. Figures 25 and 26 show the decision flow diagrams for loading a view to and from memory, respectively.

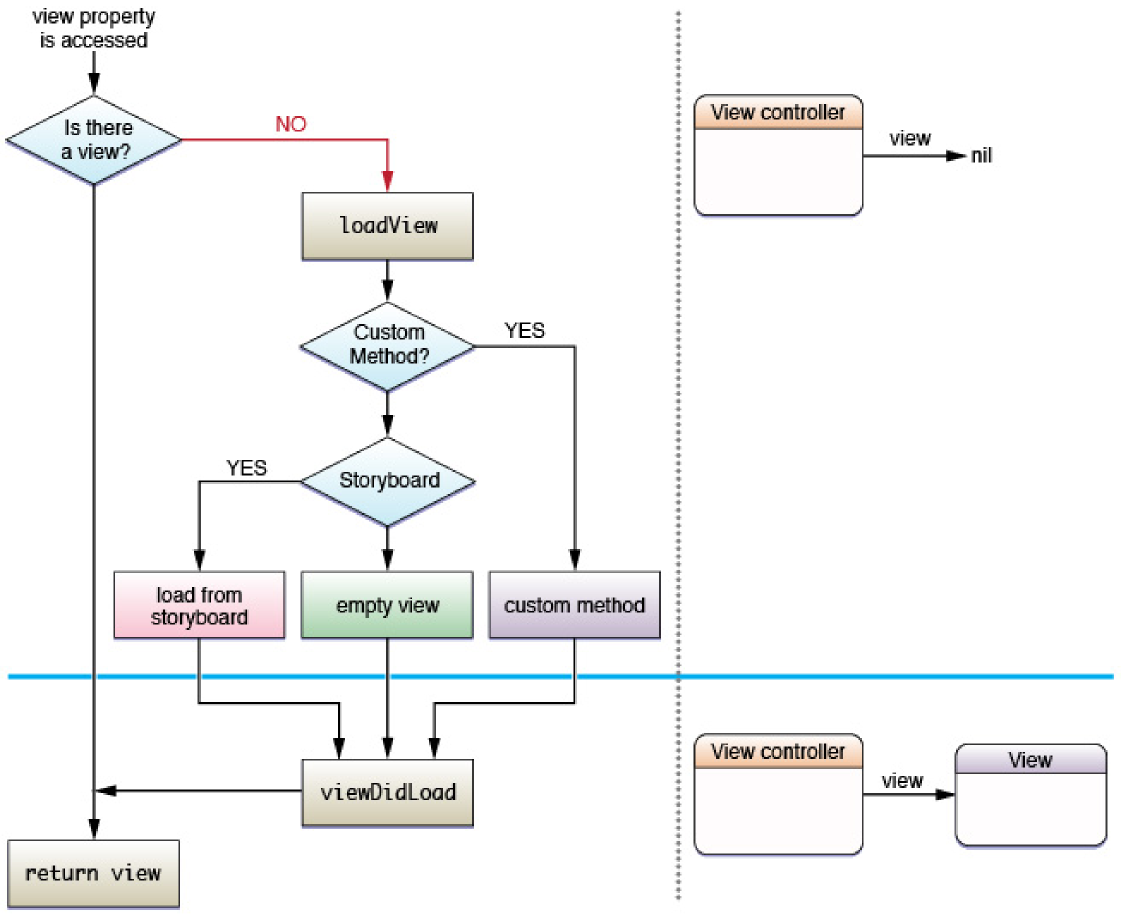


Figure 25.    Loading a View into Memory (From [16])

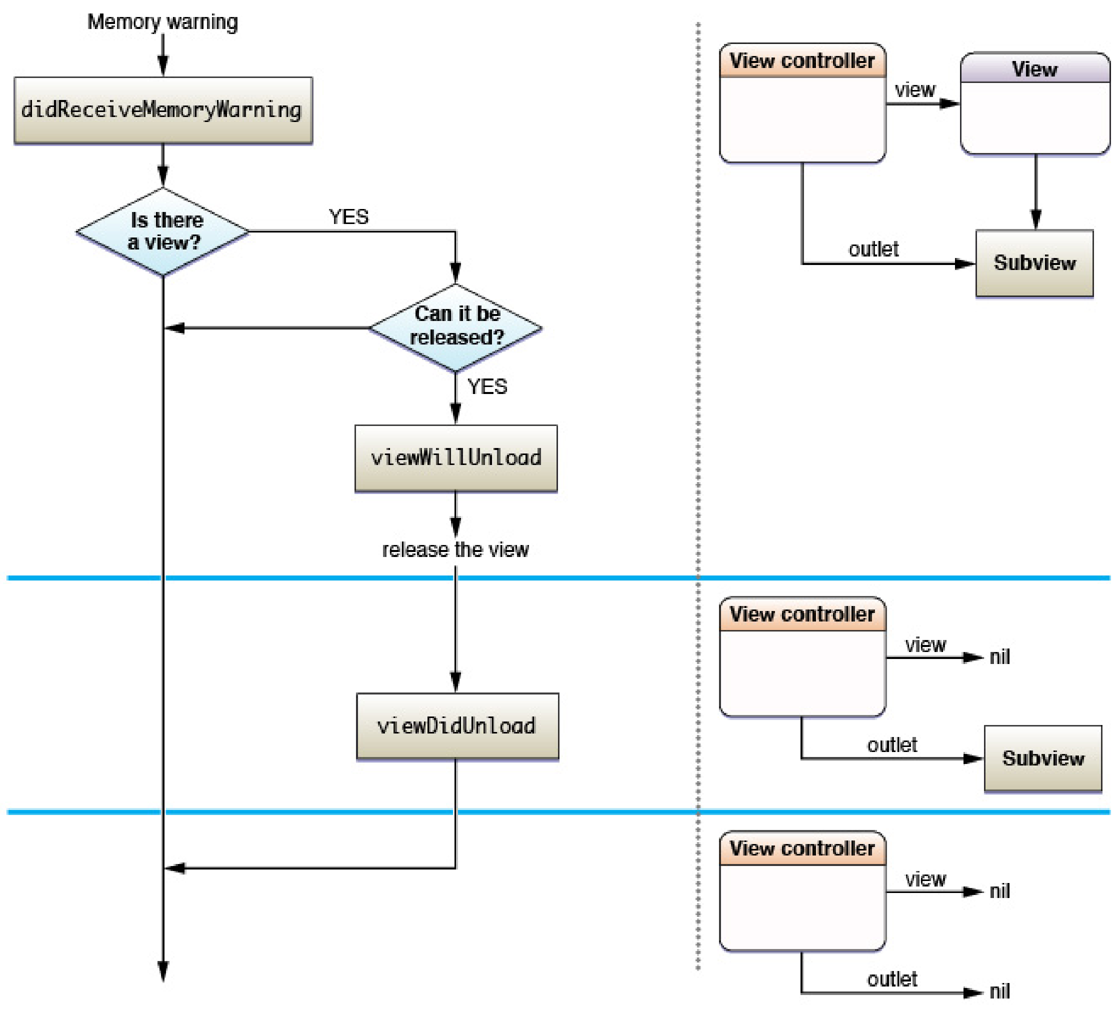


Figure 26. Unloading a View from Memory (From [16])

### 2. ViewController Lifecycle

ViewControllers come in many styles, but each has the responsibility of controlling the information flow, to and from the Views and Models, maintaining screen logic, and handling lifecycle events. These lifcycle events can be classified as: Creation, Viewing, Action Handling, Hiding, and Deallocation.

In iOS each controller has default methods to handle these events, but as an application is developed the functions can be customized to create a more dynamic experience. The most changed lifecycle functions are: init, viewDidLoad, and viewDidUnload.

## E. CONCLUSION

Patterns, in general, and MVC in specific, provide a more abstract approach with which to design applications targeting multiple platforms. By using patterns in early stages of design, developers can more easily approach specific design issues on multiple platforms with less redundant effort. In our Heart Beat Application, we were able to recognize that the Heart class could act as a model and was easily implemented on both platforms. The Views were also easily implemented on both platforms, providing guidelines to follow, but allowing us the flexibility to tailor the UI to the norms of each platform. The Controller can implement similar functionality between platforms, tying the Model to the View, but it is a perfect place to add platform specific functionality and account for platform differences.

In order to see how these patterns apply to our design algorithm and mobile programming in general, a basic understanding of generic navigation and user interface objects from Chapter III is essential.

# V. ANDROID DEVELOPMENT

This section details the Android development process and the implementation of the Model View Controller (MVC) Design Pattern in Android applications. The Android Operating System does not rely on the MVC Pattern, but the pattern can be realized in any application. Developers who have an understanding of the MVC pattern can apply it to applications they design. The following sections will highlight how the MVC Pattern may be applied to Android applications and how Android OS handles application and activity lifecycles.

## A. INTEGRATED DEVELOPMENT ENVIRONMENT

Android applications can be developed using any IDE that supports the java language, however, Google recommends and supports the Eclipse IDE. Google has developed the Android Software Development Kit (SDK) and Android Development Tools (ADT) that integrate into Eclipse providing a more seamless Android development experience. The Android SDK and ADT provide the necessary tools, APIs, and plug-ins to design, create, and test applications. The Eclipse IDE was used to develop all projects related to this thesis.

When starting an Android project you are provided limited setup options, such as: package naming and target platform (API Level). By choosing a lower API Level developers can ensure the maximum compatibility with multiple platforms. However, applications that target lower APIs have decreased support for newer functionality inherent in later APIs and platforms. Choosing a higher API

Level provides access to more advanced functionality, but results in the exclusion of earlier Android versions.

New Android projects, by default, provide developers a “main” Activity, which consists of .java and .xml files. This activity will correlate to a screen, as defined in Chapter III, and will be the default activity opened when the application is first opened. Android applications will typically consist of multiple, loosely coupled activities; each capable of opening other Activities, both internal and external to the application, through a method called an Intent. Intents allow Android applications to have multiple entry-points as well as the ability to incorporate functionality included in other application activities.

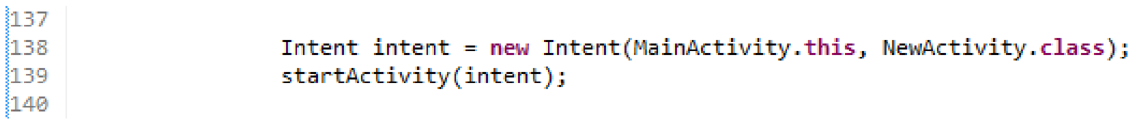
## B. NAVIGATION

Android navigation centers on the concepts of Tasks and the Back Stack. Android developer documentation defines a task as “a collection of activities that users interact with when performing a certain job” [17]. This implies that developers should identify both an overall task and a collection of sub-tasks that they expect their users to accomplish with their app. The sub-tasks should be tied to individual screens or “Activities.” These activities are organized in a stack, called the “Back Stack,” in the order in which they were opened. This aligns very well with the general navigation concepts discussed in Chapter III. The following sections provide a description of how Android implements the navigation concepts described in that Chapter.

### 1. Stack Navigation

Android implements stack navigation through the use of the Back Stack. The home screen, or Main Activity, is typically the starting place for most tasks. When an application icon is clicked, either the application's task is brought to the foreground, or if it has not been opened recently, a new task is created and the "main" activity is opened as the root of the task. When the current Activity opens another one, the new Activity is placed on the top of the Back Stack in typical last-on, first-off manner.

The ability to "push" activities onto the Back Stack is realized through the use of Intents. An Intent object is a bundle of information pertinent to the receiving component, as well as information pertinent to the Android system. Each activity can start a second activity at any time by creating an intent (Figure 27), which identifies the activity to be started and any data to be passed to that activity, and passing it into the startActivity() method.



```
            Intent intent = new Intent(MainActivity.this, NewActivity.class);
            startActivity(intent);
```

Figure 27. Android Intent

As the second Activity is started the first Activity is stopped and its state is saved. The Back Stack is never rearranged, if multiple activities are capable of opening another activity, multiple instances of that activity will be placed on the Back Stack.

"Pop" functionality is realized through the use of the back button that is a mandatory static feature of every Android device. Recent versions of Android have replaced the hardware back button with an on screen version with the same functionality

Figure 28 shows a simple application that implements stack navigation. Each of the three buttons on the main screen will open a new Activity and push it onto the Back Stack. In order to pop the new Activity from the Back Stack, a user must press the physical back button on the device.

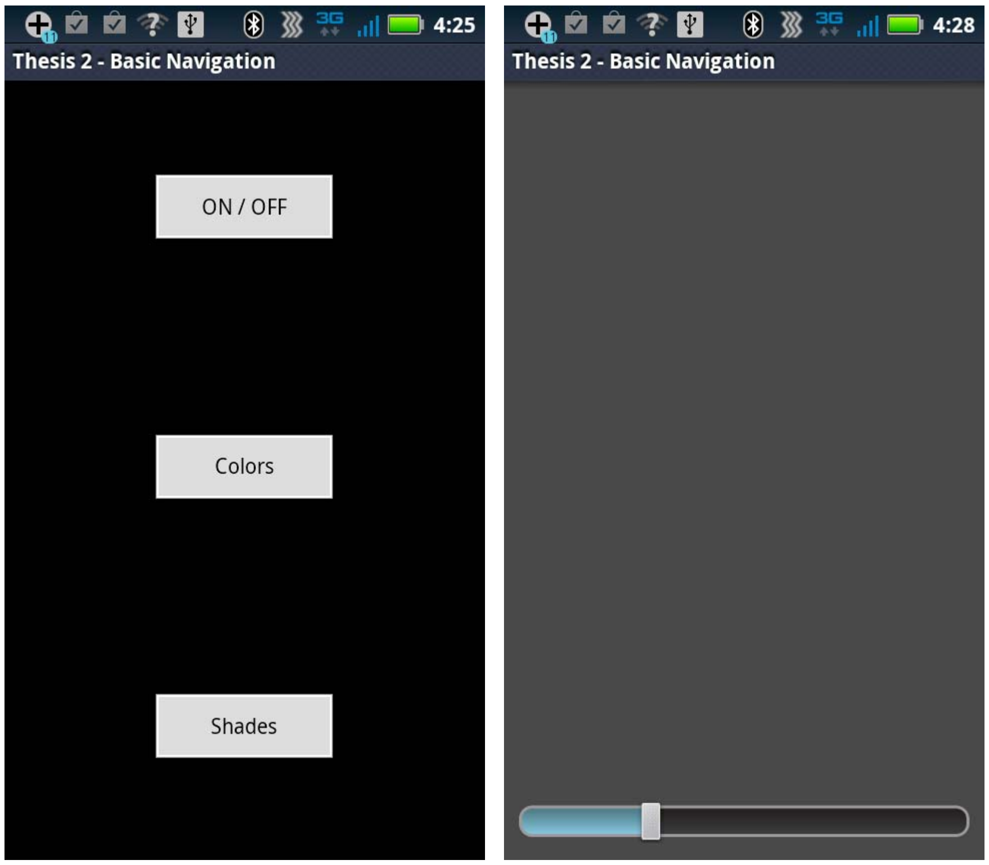


Figure 28. Android Stack Navigation Screenshots

### 2. Horizontal Navigation

Horizontal Navigation is realized by implementing a subclass of TabActivity. TabActivity is responsible for switching the displayed content when a user selects different tabs. TabActivity is a subclass of Activity and as such must include both a .java and .xml file to display the view element of the activity. TabActivity contains an instance of a TabHost, which is a container that holds multiple child activities and labels of the desired tabs. It also contains a frame layout that is used to display contents associated with a specific tab; this is typically another activity.

The child activities are responsible for all actions performed inside the frame layout specified by the TabHost. The TabHost retains responsibility for switching between children as necessary.

### 3. Composite Navigation

Due to the capabilities that Intents provide to each activity, the ability to push any other activity onto the Back Stack, Composite Navigation is very similar to Stack Navigation. The Main Activity of an Android application may be a TabActivity or an Activity. Activities may push subclasses of Activity, including TabActivity objects, onto the Back Stack. The child activities of a TabHost may push other Activity and TabActivity objects onto the Back Stack when they are active.

The major difference between Android and iOS implementation of Composite Navigation is that, by default, Android pushes new Activity objects on top of the

TabActivity vice on top of the child activity. This results in the tabs being hidden when a user navigates deeper into a child activity's hierarchy. This functionality is hown in Figure 29. This provides users with larger screen real estate and limits their focus on a single branch of their task. By default, the Back Stack is not designed to branch. This functionality must be taken into account when designing cross-platform applications.

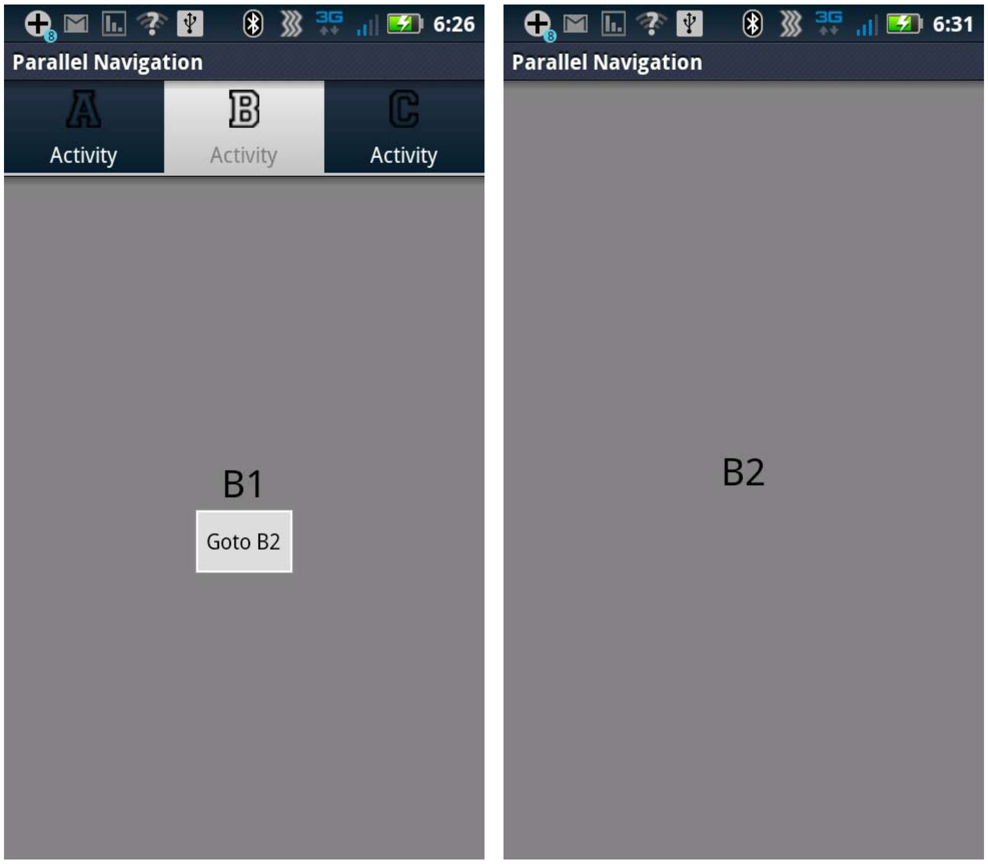


Figure 29. Android Parallel Navigation Screenshots

## C. MODEL VIEW CONTROLLER PATTERN

Android documentation does not enforce the MVC pattern as emphatically as does iOS documentation. Activities are designed to be self-contained and provide everything a developer needs to present a single screen to a user and accept feedback from the user interactions with that screen. In an effort to standardize design implementation we sought a method that we can comfortably apply the MVC pattern to Activities in an intuitive manner that aligns with current Android design paradigms.

We began our study by decomposing an Activity into its base components. The View component is described by an activity’s associated .xml document; the .xml is not, in and of itself, a View, rather a description of a composite View that will be loaded by the Controller. Model objects are even simpler to identify, all application data and logic should be stored in Model objects. Model objects may be duplicated almost entirely between platforms, developers must only be aware of minor language differences. The Activity object is the logical object to assume the role of Controller: it holds references to Models and Views, and is typically expected to provide logic associated with the Activity.

The following sub-sections will describe how to apply the Model, View, Controller pattern to an Android Activity. We will continue to utilize the Heart Beat Application to describe these concepts.

### 1. Model

Model objects in Android function almost exactly as they do in iOS with the exception of minor language differences. The Android version of our Heart Beat application implements the same Heart model as its iOS counterpart. Figure 30 shows the Android implementation of the Heart model. There are obvious language differences; Timers in Android utilize milliseconds and Timers in iOS utilize seconds, this can lead to some minor differences when actually implementing a class. Overlooking these minor differences, however, can lead to significant execution differences.

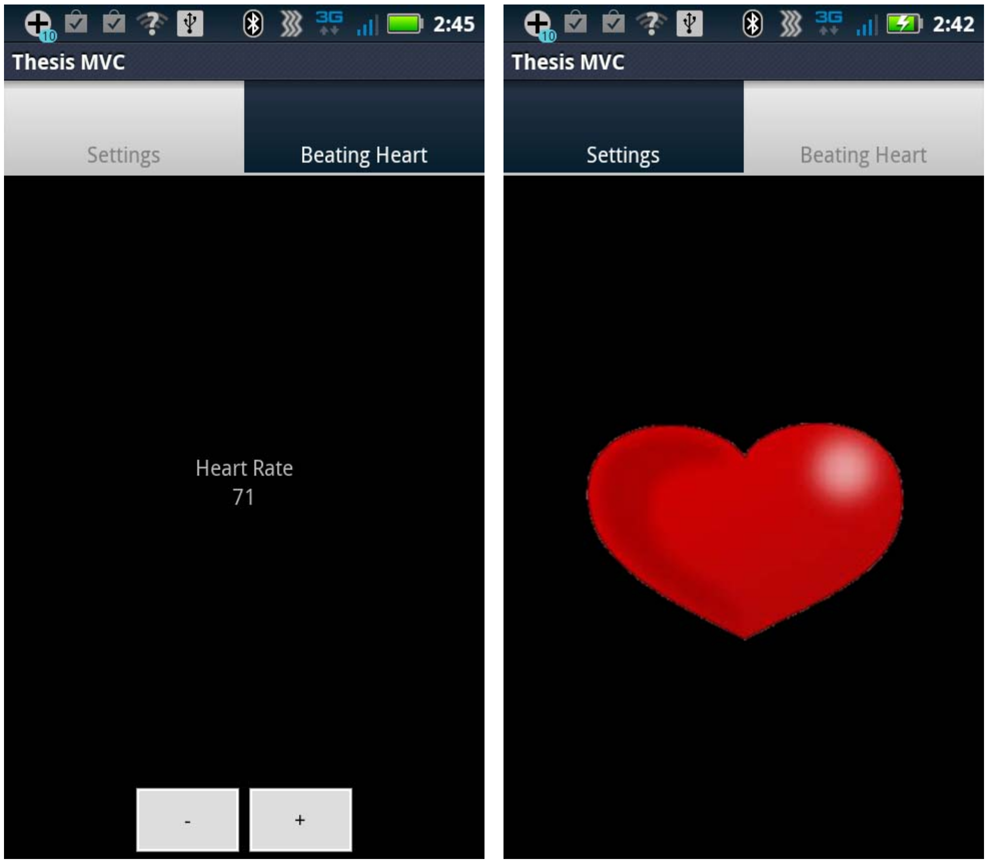
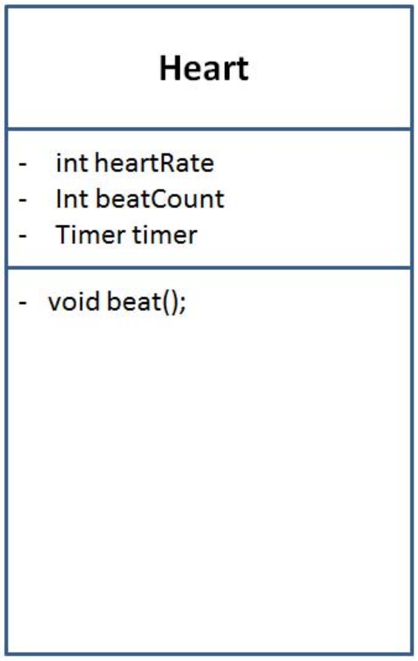
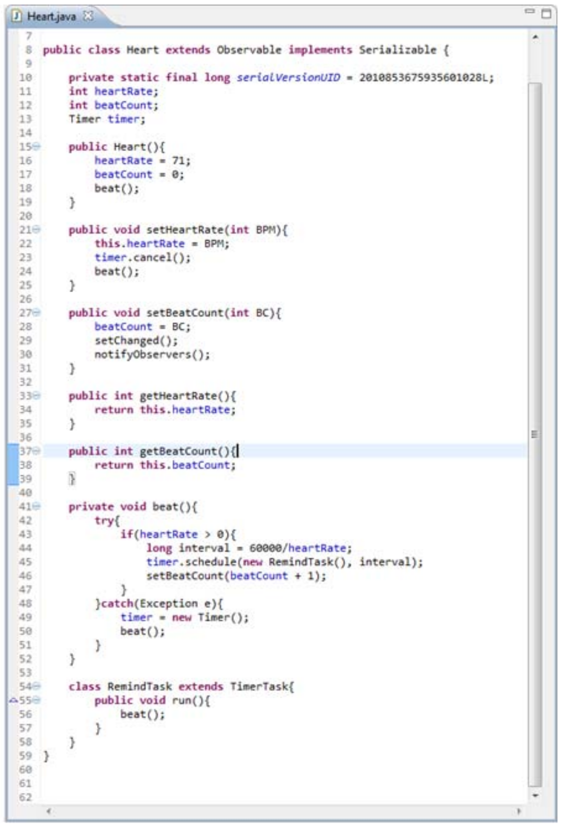


Figure 30. Heart Beat Android Screen Shots

Regardless of the minor differences, the overall structure of the class will remain constant; the same function will act in a similar manner returning the same results, regardless of the actual code utilized inside the functions. It is very easy to store application data and logic inside these models. Utilizing a common design document, functions may be easily implemented in either platform either in parallel, or by writing for one platform and then migrating the code to the second platform. Either method allows developers to maintain a common structure between platforms that will allow for easier maintenance later in the application life cycle. Figure 31 shows the design document and corresponding Android .java file.





```java

public class Heart extends Observable implements Serializable {

    private static final long serialVersionUID = 2010853675935601028L;
    int heartRate;
    int beatCount;
    Timer timer;

    public Heart(){
        heartRate = 71;
        beatCount = 0;
        beat();
    }

    public void setHeartRate(int BPM){
        this.heartRate = BPM;
        timer.cancel();
        beat();
    }

    public void setBeatCount(int BC){
        beatCount = BC;
        setChanged();
        notifyObservers();
    }

    public int getHeartRate(){
        return this.heartRate;
    }

    public int getBeatCount(){
        return this.beatCount;
    }

    private void beat(){
        try{
            if(heartRate > 0){
                long interval = 60000/heartRate;
                timer.schedule(new RemindTask(), interval);
                setBeatCount(beatCount + 1);
            }
        }catch(Exception e){
            timer = new Timer();
            beat();
        }
    }

    class RemindTask extends TimerTask{
        public void run(){
            beat();
        }
    }
}
```

Figure 31. Android Heart Model

**2. View**

As in iOS, Android View elements are organized following the Composite design pattern. Android developers may declare an activity’s layout in two ways: declaring UI elements in XML and instantiating layout elements at runtime. Android provides an XML vocabulary and simple WYSIWYG visual builder that developers may use to design their UI layouts. Additionally, View objects may be created and manipulated programmatically.

Declaring the UI in XML provides Android with the ability to support multiple screen sizes. Multiple layout documents may be created that correspond to multiple screen sizes. The appropriate layout will automatically be chosen when the activity is started.

Unlike iOS, each Activity is assigned its own root view element, typically a LinearLayout that is used to arrange child View objects in a single column or row. In addition to being ignorant of Model objects, Android View objects are also completely ignorant of the existence of their corresponding controllers. Android View objects do not call functions in their controllers when the user interacts with them. It is the responsibility of the Controller objects to listen and respond to UI events.

**3. Controller**

The Activity object is the core of each screen in an Android application. References to both Model objects and View objects are instantiated, maintained, and manipulated by the Activity objects. The Activity is the logical object to assume the role as Controller.

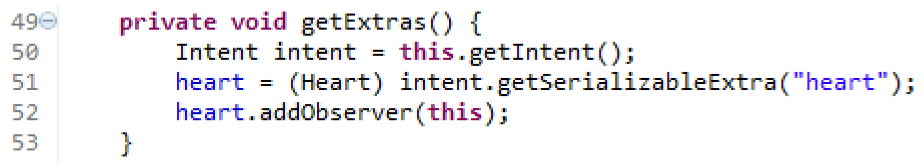
Android Controller objects will instantiate View objects by either loading an XML layout file or dynamically creating new objects and placing them into an existing ViewGroup. In the former process, the view objects, such as a Button, are assigned an ID attribute in the XML document and are referenced by the Controller based on that ID. The Controller will then instantiate an appropriate object and connect it to the item in the XML document using the assigned name attribute. In order to maintain code readability and have a logical break down of code, we created a setConnections method that links all the UI Elements in each controller.

The latter method provides developers the capability to manipulate the UI at runtime, adding additional widgets or removing them as required by the application. For the purposes of this thesis, we will rely on static UIs and will not employ the latter method.

Android Activities are not designed to function specifically as Controllers. In an effort to ensure similar code concepts between the platforms, we decided to enforce the Controller role on the Android Activity object, mirroring the functionality of the iOS ViewController. In order to facilitate this concept, we created a code structure to be used with Activity objects that attempts to align the core functionality of the two objects. We have created several functions that, when used to format an Android Activity will help maintain consistency when applying our overall design.

The Beating Heart application for Android shares a common design with the iOS Version. The screens are similar in appearance and functionality, with the exception of certain platform specific UI elements, such as tab position. The underlying code and concepts operate similarly, and the Activity objects share similar methods with the iOS ViewController objects.

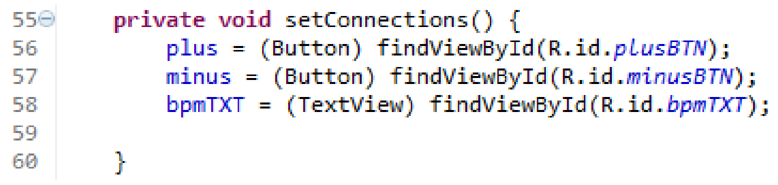
The getExtras() method (Figure 32) is used to receive information passed between screens. In the Beating Heart application, the heart model is instantiated in the TabBar Activity and a reference is passed to both child activities through an Intent. The child activities both receive the Intent and register as observers of the heart model.



```
    private void getExtras() {
        Intent intent = this.getIntent();
        heart = (Heart) intent.getSerializableExtra("heart");
        heart.addObserver(this);
    }
```

Figure 32. Android Get Extra Function

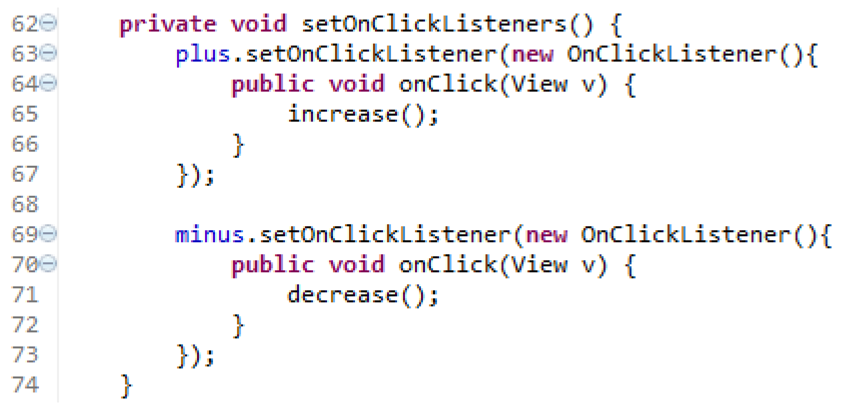
We utilize the setConnections() method (Figure 33) to connect view objects to controller reference variables. This is similar to the declaration of IBOutlets in the iOS Header files and the drag-and-drop connection of those outlets to objects in the Interface Builder. After the XML layout document is loaded, the setConnections() method will connect the view object references in the XML layout document to local variables in the Controller object.



```
    private void setConnections() {
        plus = (Button) findViewById(R.id.plusBTN);
        minus = (Button) findViewById(R.id.minusBTN);
        bpmTXT = (TextView) findViewById(R.id.bpmTXT);

    }
```

Figure 33. Android setConnections Method

The setOnClickListeners() method (Figure 34) is used to perform a role similar to connecting IBActions to individual View objects in the iOS Interface Builder. After connecting the Controller to necessary View objects, the setOnClickListener() method will fill the role of selecting target actions. Unlike iOS, Android Widgets are not associated with target actions; they are ignorant as to the existence of their associated Controllers. Android View objects simply provide a callback method by which other objects may be alerted to status changes; in the case of the Heart Beat application, the Activity acting as a Controller listens for the callback method of the associated View object. In order to simplify the relation between Activities and iOS ViewControllers, we implement the same methods in both classes. Where in iOS the plus Button would be associated with the target action increase(), the Android Button is only aware of a click occurring, the Activity that has implemented an onClickListener for that button associates the click event with the increase() method.



```
    private void setOnClickListeners() {
        plus.setOnClickListener(new OnClickListener(){
            public void onClick(View v) {
                increase();
            }
        });

        minus.setOnClickListener(new OnClickListener(){
            public void onClick(View v) {
                decrease();
            }
        });
    }
```

Figure 34. Android setOnClickListeners Method

Implementing the Activity object in this manner allows it to assume the role of the Controller. The Activity will instantiate and manipulate Model objects, control the presentation of View elements and listen for user events through onClickListener() or equivalent methods. This method provides distinct separation of Model and View roles; these objects are never aware of one another.

When the MVC is utilized in this manner in Android, one may begin to see the similarity between the Android Activity / XML layout and iOS ViewController / nib file combination. The majority of application logic can be pushed into Model objects, which, other than specific language implementations, will remain largely unchanged between platforms. View objects may be arranged and formatted according to platforms' specific requirements. Controllers may be written in such a way to compensate for minor differences in the UI implementations.

## D. LIFECYCLES

### 1. Application

Android implements the *principle of least privilege*; each application lives in its own security sandbox. Android applications run in their own Linux process, isolated by default from all other applications. Android will start a new process whenever any application component is executed and then shuts it down when no longer needed, or when in a low memory condition.

The application's lifecycle is not directly controlled by the application; instead it is controlled by the system based on the system’s knowledge of running components, the priority of those components, and the overall availability of system memory.

### 2. Activity

Each Activity may exist in one of three states: Resumed, Paused, or Stopped. An Activity that is in the Resumed state is running, is in the foreground, and has user focus. An Activity that is paused is no longer in the foreground, and does not have focus, but is still partially visible because the Activity that is in the foreground does not completely obscure it. An Activity that is Stopped is completely obscured by another activity. Activities that are in the Paused or Stopped state are still retained in memory and maintain all state and member information, but may be killed by the system in low memory situations.

In order to control how an Application handles the transition between these states, each Application must implement certain lifecycle callbacks. Figure 35 shows the Android activity lifecycle. By default, a new Subclass of Activity will override the onCreate() method and provide a hook to choose the XML layout that will be associated with that activity. All other lifecycle callbacks must be specifically overridden to provide custom features to the Activity.

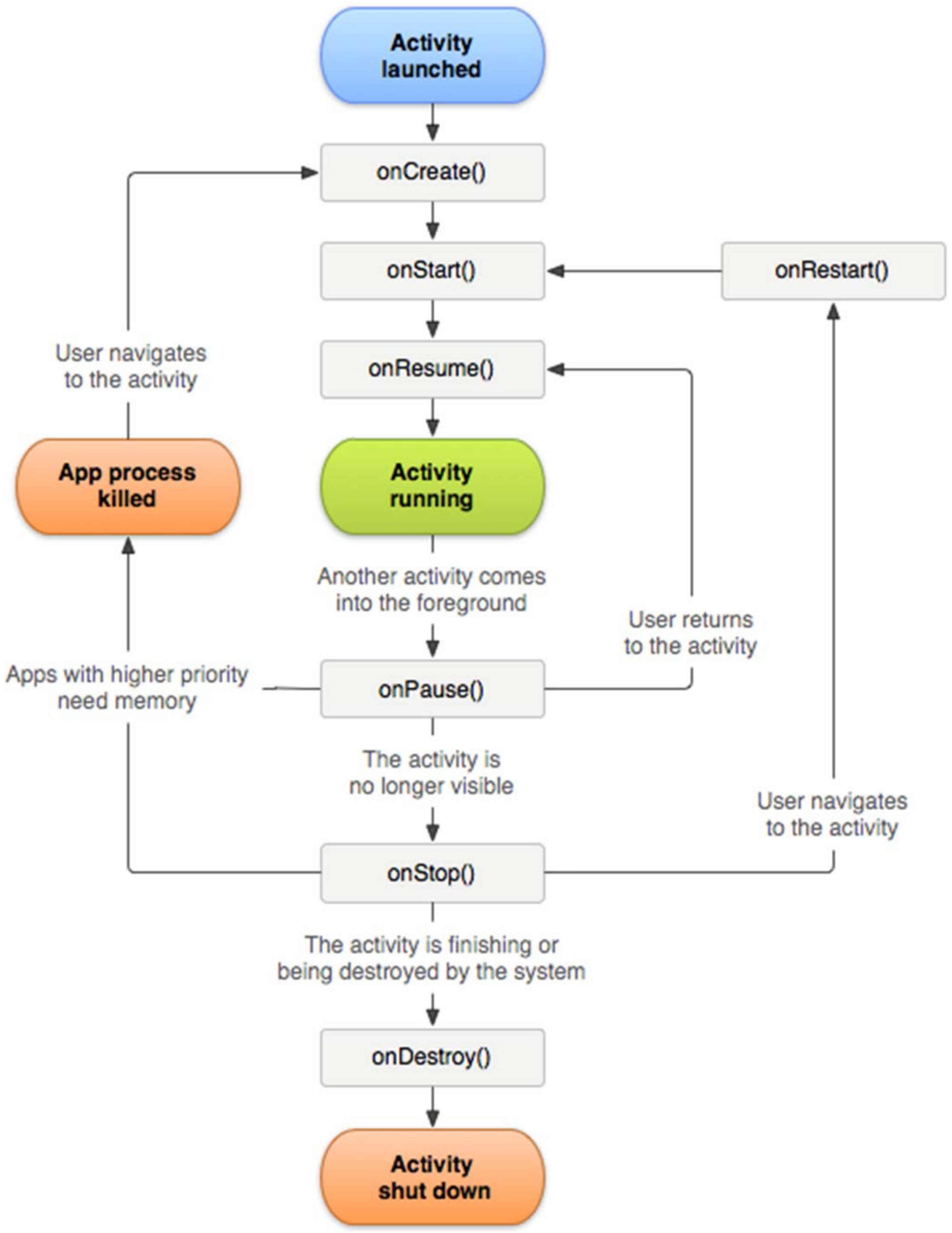

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 35. Application Lifecycles (From [18])

***a. onCreate***

The onCreate() method is called when an activity is first created; it should be used for static setup, such as binding views and instantiating or reading data received from other Activities. In the Beating Heart application, we call the getExtras() and setConnections() methods in the onCreate() call.

***b. onRestart***

The onRestart() method is only called after the Activity has been stopped and just prior to being started again.

***c. onStart***

The onStart() method is the last callback before the activity becomes visible to the user.

***d. onResume***

The onResume() method is called when the Activity is at the top of the activity stack and is visible. When it returns, the user will be able to interact with the activity.

***e. onPause***

The onPause() method is called when the system is about to start or resume another activity. It is used to store unsaved changes to persistent data and stop animations and other CPU intensive activities to persistent data. This method is the last lifecycle callback that an activity is guaranteed to have called before the application can be killed by the system. It is important to ensure that this method runs quickly, as it will block new activities from becoming active until it completes, thus affecting overall user experience.

#### *f. onStop*

The onStop() method is called after the activity is no longer visible, either because it is being destroyed or because another activity is completely obscuring it from view. This method is not guaranteed to be called, so it should not be used to store critical data or state, but may be used to perform certain non-essential shutdown tasks.

#### *g. onDestroy*

The onDestroy() method is called prior to the activity being destroyed. It is the final call the activity will receive. As in the onStop() method, this method is not guaranteed to be called, so actions should be limited to non-essential tasks.

#### *h. onSaveInstanceState*

Activities that are moved to the background maintain state in one of two ways: they are retained in memory, and thus never lose state, or they are destroyed and recreated in which they must restore a previously saved state. The onSaveInstanceState() method is called before the activity is eligible for destruction, typically before onStop(), sometimes before onPause(). This method, by default, will save the state of the various UI elements as long as they have been assigned a unique identifier, typically within the XML layout document. The default functionality will not save model data, and thus must be overridden in order to ensure that any critical model state remains intact as an activity is destroyed and recreated. It is important to note that this method is not guaranteed to be called, so the storage of persistent data, such as data stored in a file or a database should be handled in the onPause() method.

## E. CONCLUSION

By understanding the similarities in navigation concepts between iOS and Android, developers will be able to identify common tasks and correlate them to individual screens that may be implemented on either platform.

Implementing the MVC pattern in Android will accentuate the similarities between iOS ViewControllers and Android Activities. Each screen in an application will consist of a single ViewController or Activity with an associated layout document containing a list of View objects.

Lifecycle callback methods are implemented slightly differently between iOS and Android; but by understanding when these methods are called, developers can maintain many similarities between the platforms.

In Chapter VI, describes a process in which applications may be designed in such a way that they may be implemented on either platform. We followed this method developing an application for Naval Postgraduate School students to read important notifications and conduct daily muster using either an iPhone or Android device. We will use this application and its associated design documents to detail the process.

THIS PAGE INTENTIONALLY LEFT BLANK

# VI. DESIGN PROCESS

We have created a design process that will enable mobile application developers to create a single set of design documents that may in turn be implemented on either the Android or the iOS platform. The process begins by breaking a set up requirements into a set up actions, or tasks, which the user will be expected to be able to accomplish, and then building those tasks into a set of individual screens that will be realized in the application. We will rely on the Model, View, Controller pattern to design individual screens, and our previous discussions on stack and horizontal navigation to design our application’s navigation flow. This chapter will detail the process that we follow, describe the design documents that we create, and show real world application of the process as we build the NPS Muster application, which could be used for NPS students to conduct daily musters, read announcements, and access other relevant student information via NPS intranet.

## A. IDENTIFY REQUIREMENTS

To begin the design process, developers must identify the requirements for any application they wish to build. The processes involved in this step may vary drastically between design teams and is not integral to this thesis. It is only important to note that at the end of this step the developer should have a reasonable understanding of the application requirements.

We envisioned the concept of our example application based on requirements that we have as students at the Naval

Postgraduate School (NPS). We are required to conduct an online muster, or check-in, every weekday. The muster is completed through the verification of a captcha image to ensure that students are not using automated programs to conduct musters on their behalf. In conjunction with that muster, we must read through a page of announcements of various importance, some of which change from day to day. We often found ourselves referencing resources from the campus intranet in conjunction with our daily musters.

From our experience with the traditional method of mustering we identified the requirements listed in Table 1. Well thought out and understood requirements provide a necessary base and starting point for our design process. The developer may not be required to identify these requirements, but should seek an active role in understanding and clarifying them with the customer.

| **Requirement** | **Comments** |
|---|---|
| 1. Provide Credentials | Users must enter username / password combination in order to use the application. |
| 2. Store Credentials | Users are given the option to store username or username and password. |
| 3. View Announcements | Users should be able to view all active announcements. |
| 4. Sort Announcements | Announcements should be sorted by priority. |
| 5. Mark Announcements as Read | Announcements that have already been read should be marked as such. |
| 6. Read All Announcements | Students are expected to read each announcement before being allowed to muster. |
| 7. Prevent Automatic Mustering | Students must not be provided the capability to automatically muster each day, they must physically conduct the muster on their device |

| | |
|---|---|
| | each day. |
| 8. Show Muster Status | Upon logging in, student should be notified of current muster status. |
| 9. Complete Daily Muster | If not currently mustered for the day, students should be allowed to conduct muster from the application. |
| 10. Provide Intranet Access | Using credentials stored from current user session, provide students access to NPS Intranet resources.<br>Note: this is a low priority requirement |

Table 1. NPS Muster Application Requirements

## B. IDENTIFY TASKS

After identifying the requirements for a new application, developers should begin reviewing the processes, or tasks, that will be required to fulfill these requirements. A task is a simple action that a user would be expected to accomplish as a step to complete a requirement. Tasks may correspond to requirements in a variety of ways. They may serve as a stepping-stone to fulfill a single requirement, multiple requirements, or as a bridge between multiple requirements. Tasks should always align with one or more requirements, otherwise the application will experience design drift, or “requirements creep,” and begin to include extraneous functionality.

One example of a task in the NPS Muster Application would be “Log in”. Users are asked to provide their individual NPS username and password combination in order to use this application. This task aligns with the Provide Credentials and Store Credentials requirements. Table 2 provides a complete list of our identified tasks for the NPS Muster Application. Listing all the tasks and the related requirements helps to ensure that the application meet all the requirements.

| Task | Requirements | Comments |
|---|---|---|
| Login | 1 and 2 | User is provided ability to enter username and password and options to store credentials for future usage. |
| Read Announcements | 3, 4, 5, and 6 | A student will be presented a list of announcements that are sorted by read status and priority. The application has limited functionality if any announcements are not marked as read. |
| Muster | 7, 8, and 9 | The user is notified of current muster status. If the user is not currently mustered, then provide a method in which the user must manually interact with the application in order to change the status to 'Mustered.' |
| Intranet Access | 10 | NPS intranet requires users to enter credentials. This Task will provide streamlined access to intranet resources using credentials stored only for this session. |

Table 2. NPS Muster Application Tasks

## C. IDENTIFY REQUIRED SCREENS

Once developers have identified the tasks which users are expected to accomplish using the application, they should focus on designing basic, platform independent, versions of individual screen UIs. These screens should focus on accomplishing the identified tasks outlined in the previous step. This process will help developers understand how they expect their users to interact with each of their screens.

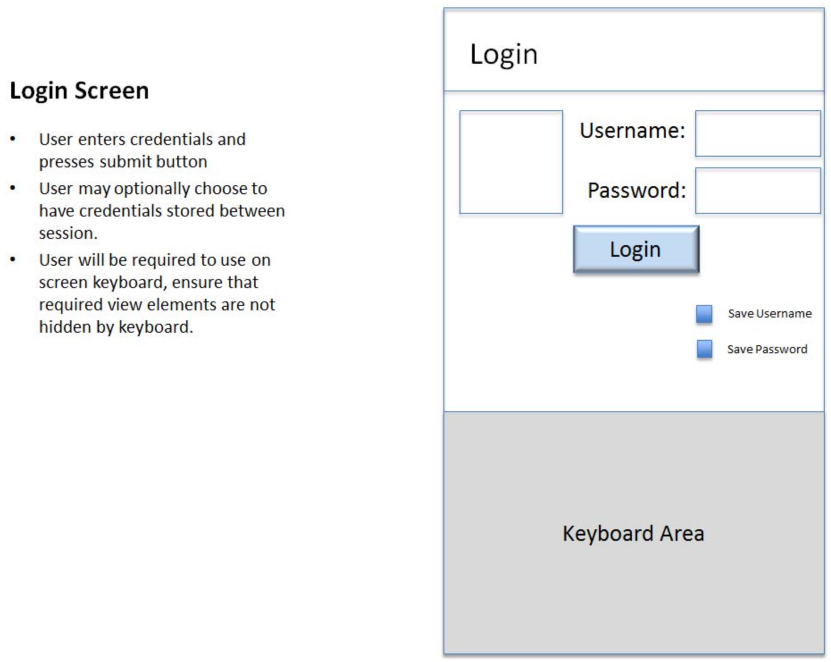
Due to limited screen size of mobile devices, many tasks may need to be broken into smaller sub-tasks. This may often result in multiple screens being required to accomplish a single task. A common occurrence of this is reading through a list of related items that may offer more detail than is presented in the list. This type of task is often handled by building a screen with a list and a screen that displays details of individual items.

Some screens may allow the completion of multiple tasks. Tasks are often so interrelated that it becomes impossible to separate them into individual screens. This is acceptable, but should be designed carefully. Too many tasks on one screen can result in a cluttered UI and user confusion. It is our recommendation that, whenever possible, a screen should be designed to accomplish a single task.

Identifying screens in this manner helps the developer modularize the code and allows for a minimalistic approach to programming. Only the code related to the interaction with the screen should appear in the controller. It also helps the developer identify and push logic code into the models. This will be discussed in detail later.

### 1. Login Screen

Using our previously identified Tasks for the NPS Muster Application, we chose to begin designing a screen based on our first task: “Log In” we chose to create a basic layout with text fields for username and password, a submit button, and check boxes, the latter allowing the user to choose whether credentials are saved between sessions. Figure 36 shows our UI design document for the login screen of the NPS Muster application.



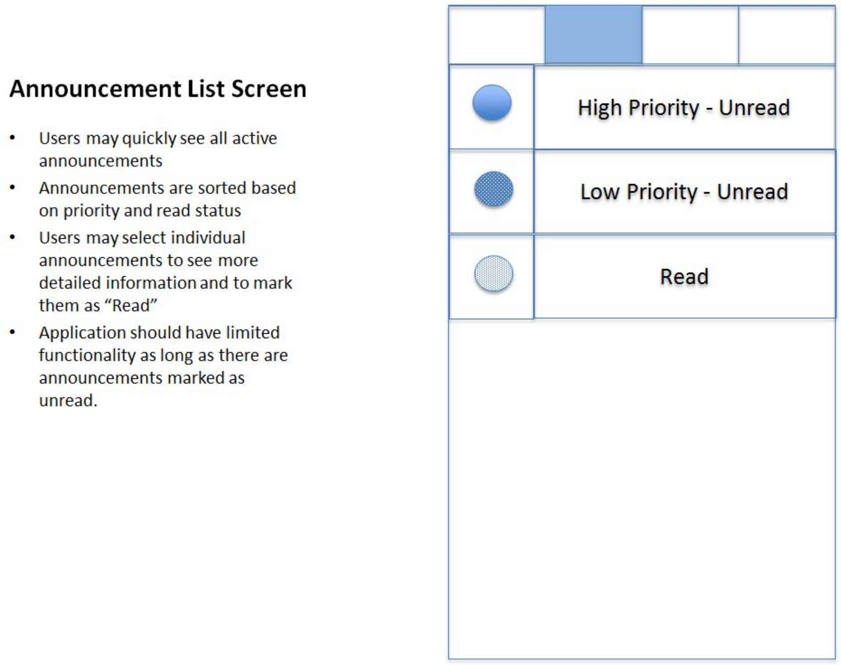
**Login Screen**

- User enters credentials and presses submit button
- User may optionally choose to have credentials stored between session.
- User will be required to use on screen keyboard, ensure that required view elements are not hidden by keyboard.

Figure 36. Login Screen Design Document

### 2. Announcement Screen

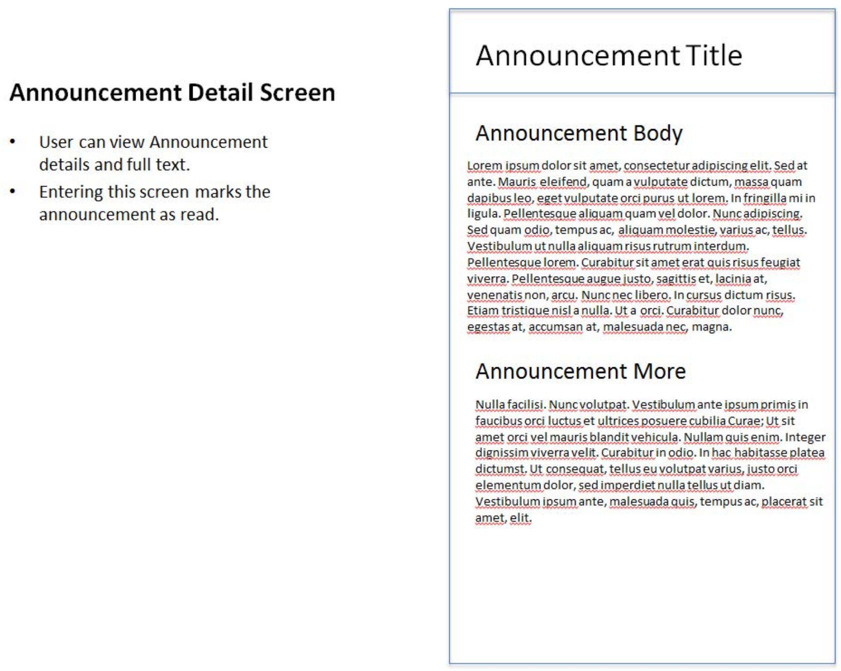
We decided that the “Read Announcements” task would best be handled by breaking the task into subtasks. There are many active announcements at any given time and we wanted to display a list of current announcements in a sorted order. This requirement is best handled by implementing a list to view the announcements. We realized that the list items are best suited to display basic information regarding each announcement, such as title, priority, and date. The details of each announcement would require a separate screen containing the body text and additional information. Figure 37 and Figure 38 describe the Announcement List screen and Announcement Details screen, respectively.



**Announcement List Screen**

- Users may quickly see all active announcements
- Announcements are sorted based on priority and read status
- Users may select individual announcements to see more detailed information and to mark them as "Read"
- Application should have limited functionality as long as there are announcements marked as unread.

Figure 37. Announcement List Design Document



**Announcement Detail Screen**

- User can view Announcement details and full text.
- Entering this screen marks the announcement as read.

Announcement Title

Announcement Body

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed at ante. Mauris eleifend, quam a vulputate dictum, massa quam dapibus leo, eget vulputate orci purus ut lorem. In fringilla mi in ligula. Pellentesque aliquam quam vel dolor. Nunc adipiscing. Sed quam odio, tempus ac, aliquam molestie, varius ac, tellus. Vestibulum ut nulla aliquam risus rutrum interdum. Pellentesque lorem. Curabitur sit amet erat quis risus feugiat viverra. Pellentesque augue justo, sagittis et, lacinia at, venenatis non, arcu. Nunc nec libero. In cursus dictum risus. Etiam tristique nisl a nulla. Ut a orci. Curabitur dolor nunc, egestas at, accumsan at, malesuada nec, magna.

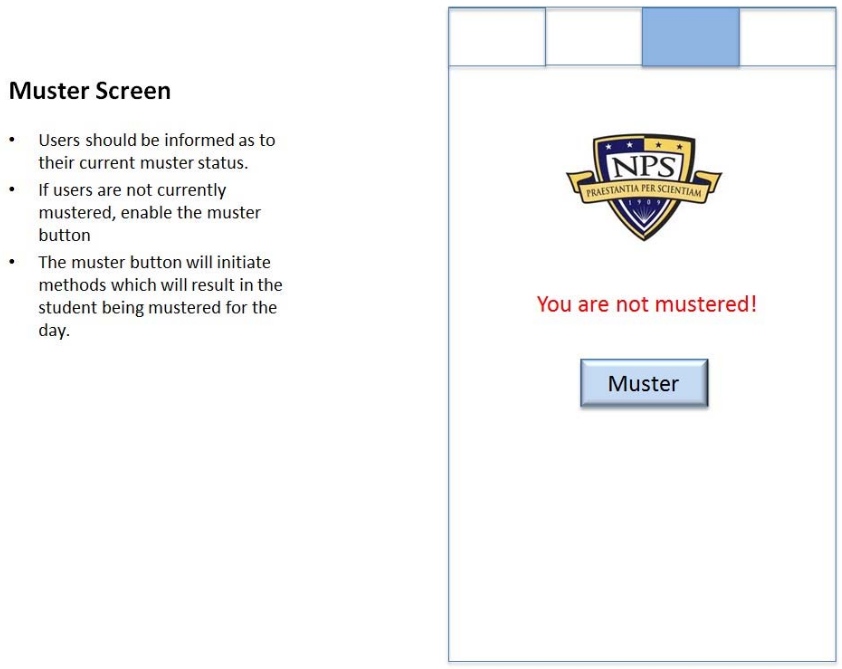
Announcement More

Nulla facilisi. Nunc volutpat. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Ut sit amet orci vel mauris blandit vehicula. Nullam quis enim. Integer dignissim viverra velit. Curabitur in odio. In hac habitasse platea dictumst. Ut consequat, tellus eu volutpat varius, justo orci elementum dolor, sed imperdiet nulla tellus ut diam. Vestibulum ipsum ante, malesuada quis, tempus ac, placerat sit amet, elit.

Figure 38. Announcement Detail Design Document

### 3. Muster Screen

The Muster task requires that we display the user’s current muster status, provide a method in which the user may complete the daily muster, and prevent the user from doing so in an automated manner. The current web version of the muster application utilizes a captcha text with which a user must verify some visual text that would be difficult for a computer to recognize. This function is necessary because without the captcha it is relatively simple to write an automated script that could complete the muster on behalf of a student. The secure nature of Android and iOS applications prevents such automatic intrusion. We chose to simplify the process by requiring that a user simply press a button that becomes active only when the user has not been mustered for the day. In future iterations of the application, if additional precautions are required, we could simply remove the functionality which allows users to save their credentials to their device, a more secure option in any case. Figure 39 shows the design document for the NPS Muster Application muster screen.



**Muster Screen**

- Users should be informed as to their current muster status.
- If users are not currently mustered, enable the muster button
- The muster button will initiate methods which will result in the student being mustered for the day.

Figure 39. Muster Screen Design Document

### 4. Intranet Screen

The intranet screen becomes a simple, yet versatile webview that will facilitate the use of other resources available on the NPS Intranet. We facilitate access by providing session credentials to access the site, but limit browser functionality to back and forward navigation. Figure 40 shows the Intranet Screen Design Document.

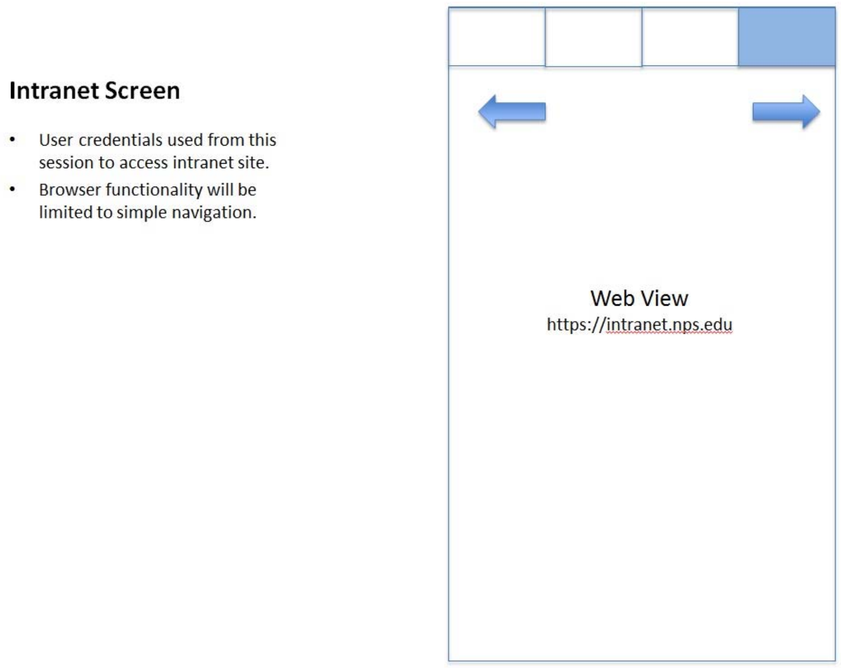

Intranet Screen
User credentials used from this session to access intranet site.
Browser functionality will be limited to simple navigation.
Web View
https://intranet.nps.edu

Figure 40. Intranet Screen Design Document

## D. DESIGN NAVIGATION HIERARCHY

After identifying the screens that will be used to accomplish the required tasks, the developer must identify how those screens will interact with one another and the order they will be shown. Understanding these interactions will drive the overall flow of the application and play an integral role in understanding how data will be shared between the screens. This section relates to earlier discussions about navigation concepts. The vast majority of applications will rely on some form of Stack Navigation, while others will utilize Horizontal Navigation. Most applications that rely on Horizontal Navigation will also include some form of Stack Navigation, resulting in the reliance upon a form of Composite Navigation structure.

### 1. Identify How Screens Will Interact

Screens should be viewed as loosely coupled and self-contained. Not all screens, however, can operate independently, and thus require some form of input arguments. In some cases, a screen may be used to accomplish a task, or sub-task, by a parent screen. This requires the capability of returning data to the parent screen. This feature is often realized in situations where a detail screen is opened by a list screen to modify an element of the list.

We found the process of actually arranging our screens into user-friendly navigation hierarchies to be somewhat challenging. Continuously redrawing flow charts proved to take a large amount of time. We began using a storyboard concept, paper printouts of our individual screens, which we were able to rearrange to simulate stepping a user through the application. This gave us a feel for how we wanted the application to flow, and began giving us insight as to what data should be passed between our screens.

For the NPS Muster Application, we found that our Login screen functioned well as a gateway, preventing unauthorized users from proceeding beyond that screen. We chose to set that screen as the start screen. Every time a user opens the application, they are shown the login screen and have to enter their credentials and manually press a button to proceed. Based on the concept of using the Login screen as a gateway we found that it made sense to implement Stack Navigation from this point, pushing the next screen on top of the Login screen.

Once a user logs-in, there are a few loosely related tasks the user could perform: Muster, View Announcements, or Access Intranet. Based on these tasks, which have been identified as screens, we decided that a Horizontal Navigation concept would work well to allow the user to quickly switch between these functional tasks. We decided to implement a tab bar that would contain each of these screens as children. Due to the nature of Horizontal Navigation and our requirement that all announcements must be read before allowing a user to muster, we identified a need for each screen in the resulting composite hierarchy to “know” the state of unread announcements.

Another reason we chose to use screen printouts as storyboards is the fluidity with which a developer may manage early prototypes of the UI flow. Developers may proceed through several versions of early prototypes, called Rapid Prototyping, and even identify additional required screens, all without writing any code. Assuming that the developer is writing an application based on a set of customer requirements, this step also facilitates communication between developers and customers by ensuring that the developer and customer agree on the final outcome of the UI flow and functionality. Once the general flow of the application has been established, it is time to begin translating it into a more formalized document.

### 2. Draw UI Navigation Diagram

Once a developer has laid out the navigation flow for an application, he or she should formalize a UI Navigation Diagram. This diagram will ensure that the navigation flow of the application is followed throughout the rest of the design process as well as providing developers a clear picture of the flow of the entire application.

The UI Navigation Diagram not only provides insight into the flow of the application, but also aids in identifying Controller and View classes that will need to be implemented in later steps. Figure 41 displays our UI Navigation Diagram for the NPS Muster Application.

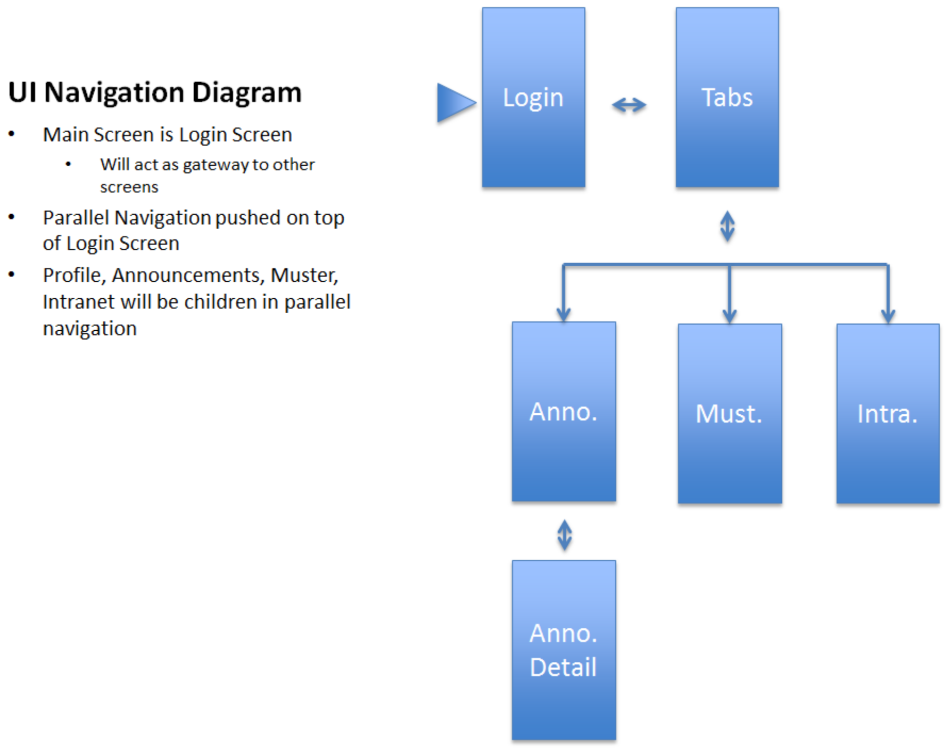

UI Navigation Diagram
Main Screen is Login Screen
Will act as gateway to other screens
Parallel Navigation pushed on top of Login Screen
Profile, Announcements, Muster, Intranet will be children in parallel navigation
Login
Tabs
Anno.
Must.
Intra.
Anno. Detail

Figure 41. UI Navigation Diagram

### 3. Draw a Data Flow Diagram

After drawing the initial UI Navigation Diagram, developers should begin mapping out the data flows that correspond to the navigation flow. While screens are mostly independent, they may require certain inputs from parent and child screens. In order to later classify the data elements that should be shared between screens, we overlaid a Data Flow Diagram on the Navigation Diagram (Figure 42). This diagram shows only the flow of required data, it does not classify the data into model objects at this point. Later, we will use this diagram to begin identifying model objects.

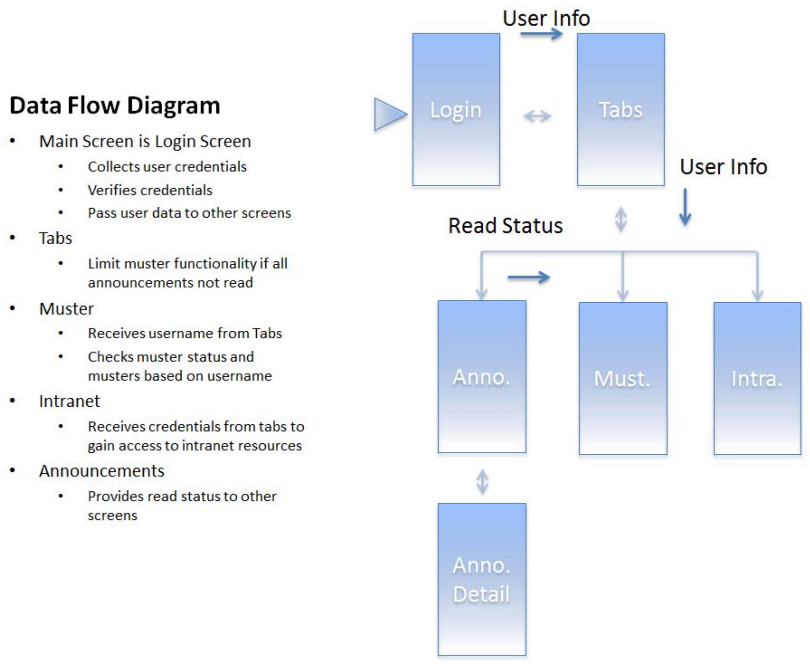

Data Flow Diagram
• Main Screen is Login Screen
• Collects user credentials
• Verifies credentials
• Pass user data to other screens
• Tabs
• Limit muster functionality if all announcements not read
• Muster
• Receives username from Tabs
• Checks muster status and musters based on username
• Intranet
• Receives credentials from tabs to gain access to intranet resources
• Announcements
• Provides read status to other screens
User Info
Login
Tabs
User Info
Read Status
Anno.
Must.
Intra.
Anno. Detail

Figure 42. Data Flow and Navigation Diagram

## E. DESIGN CLASSES

After visualizing the overall flow of the application, both in terms of screen navigation and data flow, it is necessary to begin identifying and designing the specific components and classes that will be used to build the screens and represent the data. The goal is to apply the MVC pattern to each screen; in order to do this, the developer should rely on the UI Navigation Diagram and the Data Flow Diagram to identify Model, View, and Controller objects. First we will use the UI Navigation Diagram to identify View and Controller objects, and then we will use the Data Flow Diagram to identify the required Model objects. Once we have identified the required classes we will focus on the intricacies of each individual class, ensuring that the models are defined in such a way that they may be implemented regardless target of platform. We will show how to design Controllers in such a way as to accentuate the similarities of target platform implementation while minimizing the differences. We seek to maintain as much similarity as possible when implementing Views, but also ensure that they align with platform design philosophies, ensuring that the user is presented with a platform specific experience, thereby conforming as much as reasonably possible to the “look-and-feel” of applications that the user expects when using the specific platform.

### 1. Models

Model objects may be written in such a way that the class variables and methods are consistent across platforms. Other than obvious language differences, models should be designed to function exactly the same, regardless of platform. Developers can expect that the controller’s interaction with a model’s methods will be the same despite the underlying code. When properly designed, the models should be viewed as black boxes, always taking correct inputs and giving expected outputs.

To begin identifying required Model objects, the developer should begin with the Data Flow Diagram. When data is passed between screens, if the data is not primitive, it should be passed as a Model. Models may exist in the scope of a single screen, never being shared with another screen; these models are much harder to identify. It should be the goal of the developer to wrap data objects into logical models with specific functions. For example, the Beating Heart application held a single Model that was shared between multiple screens in order to accentuate the concept of the MVC pattern. Not every application is that simple; there may exist any number of Models associated with any number of screens. Model objects may actually be composed of several other models.

In order to simplify this discussion we sought to utilize only one model object that would hold all of the required data and provide the majority of the required logic for the NPS Muster Application.

#### *a. Profile*

We chose to create a Profile class to act as a single, overarching model for the entire application. This class would contain all functionality and data required throughout the application. It is in fact, a composite model, as it would also contain other model objects for use in different screens. The model would initially be created the first time the user logged in, passed to various screens, and subsequently stored for future sessions. This class would contain all data and logic required by the user to verify and store credentials, review announcements, muster, and access intranet resources. Figure 43 shows the class diagram for the Profile class.

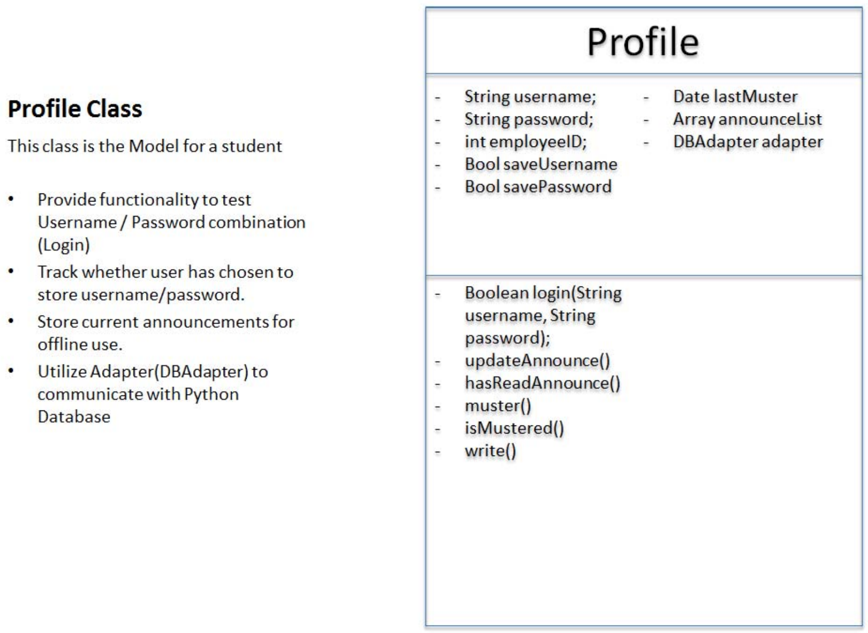

Profile Class
This class is the Model for a student
Provide functionality to test Username / Password combination (Login)
Track whether user has chosen to store username/password.
Store current announcements for offline use.
Utilize Adapter(DBAdapter) to communicate with Python Database
Profile
- String username;
- String password;
- int employeeID;
- Bool saveUsername
- Bool savePassword
- Date lastMuster
- Array announceList
- DBAdapter adapter
- Boolean login(String username, String password);
- updateAnnounce()
- hasReadAnnounce()
- muster()
- isMustered()
- write()

Figure 43. Profile Class Diagram

***b. Announcement***

The Announcement class will be used to contain the data and associated logic of a single announcement. These model objects will be contained in an array held by the Profile. The Announcement model acts mostly as a data container, but contains logic required to compare itself to the online version of itself to determine whether it has been updated on the remote database. Announcements will be synchronized to the device and updated when the user logs-in. The model objects will contain an algorithm that compares the local version of the announcement to the version stored in a remote server. As long as no changes are present, then the announcement will retain its read status, if changes are present, the announcement will update itself displaying a not read status. Figure 44 shows the class diagram for Announcement.

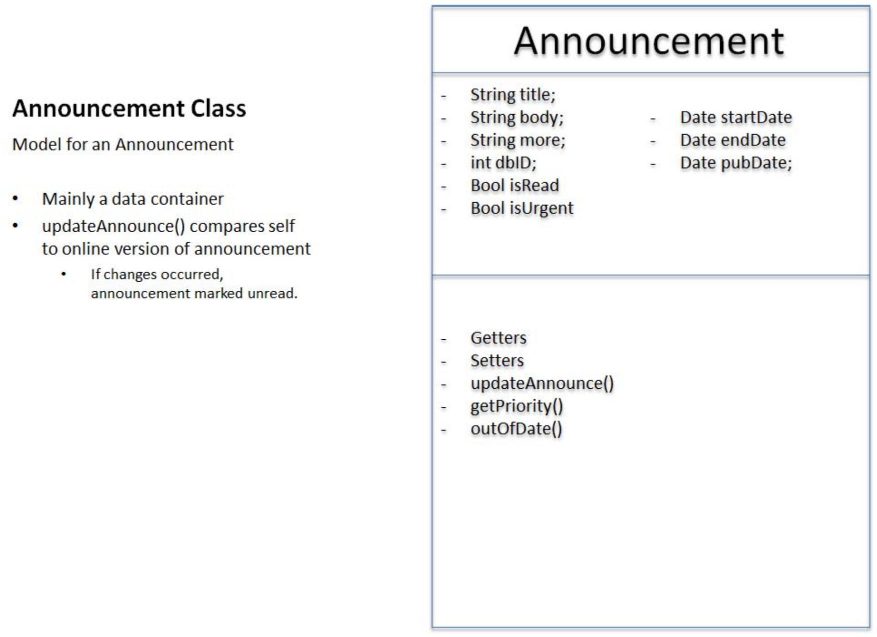


Figure 44. Announcement Class Diagram

### 2. Views

Views follow the Composite design pattern; when organized into hierarchies, groups of views become a composite view or a view layout. We chose not to rely on class diagrams to represent our views layouts. In our final step, implementation, we utilized the WYSIWYG editors included in Xcode and Eclipse to design platform specific view hierarchies. Using the included interface editors ensures that the developer is creating a UI that aligns with the suggested guidelines set forth by the platform designers.

### 3. Controllers

Each screen consists of a Controller and an associated View hierarchy. Each screen requires a Controller class to be designed for it. Controllers are implemented differently between iOS and Android platforms, and thus present much more variation between platform implementations than Model objects. We mitigated these differences by identifying three functional areas common to Controllers in either platform. As such, each Controller consists of a Lifecycle, an Interface, and a Logic section. The Lifecycle allows the controller to determine when the screen it is controlling is active, when it is in the foreground or background, or when it is being removed. The Interface determines how the Controller will connect to, and interact with, the associated View hierarchy. The Logic section determines how the Controller interacts with, and modifies, any associated Models.

We first utilized a separate class diagram for each platform when designing our Controllers. Later we were able to merge these diagrams, but it is essential to understand the concept of the separate documents before utilizing the unified diagram. Figure 45 presents the separate class diagram for Android and iOS Login Controllers.

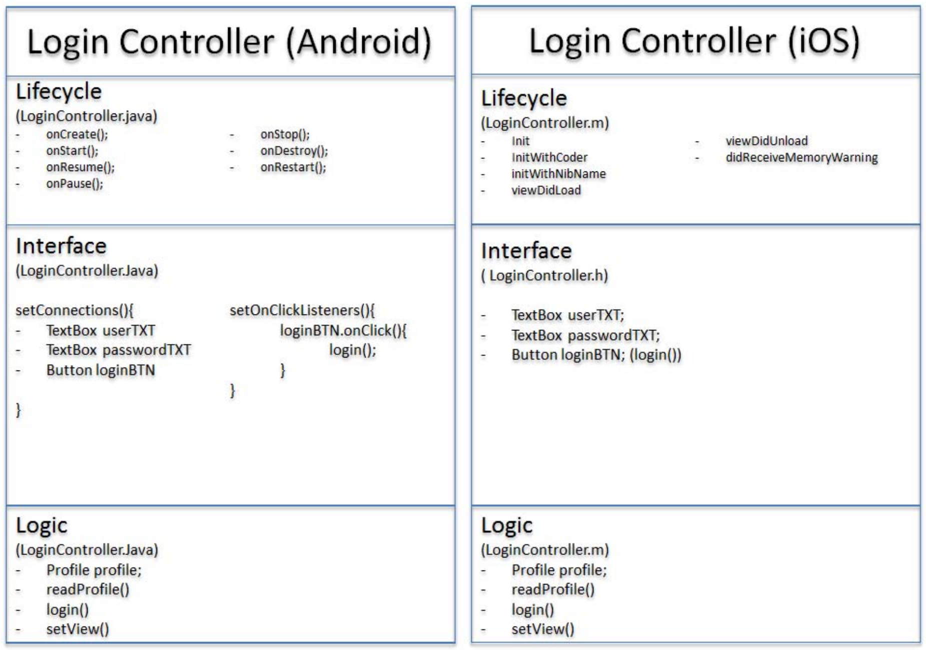


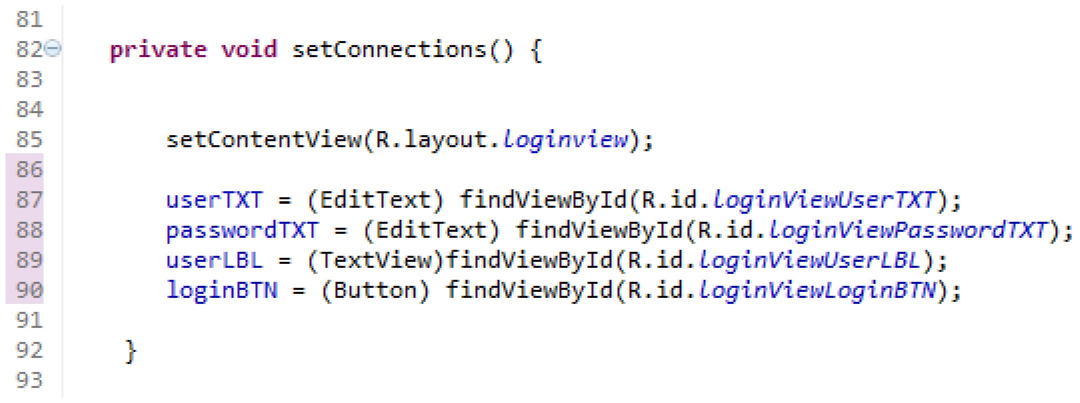
Figure 45. Separated Login Controller Diagrams

***a. Lifecycles***

The lifecycles of Android activities and iOS controllers are noticeably different. Android relies heavily on the concept of Activity lifecycles where iOS provides simple callbacks that indicate when the controller is initialized, loads, or unloads. While the actual lifecycles are rather different, the underlying functionality is relatively similar. The Android onCreate() method equates to the iOS Init methods and should be used for completing setup actions prior to presenting the screen to the user. The onResume() method may be equated to the iOS viewDidLoad() method and is called immediately after the screen becomes visible to the user. The onPause() method, being the last method an Android activity is guaranteed to receive, should be equated to the viewDidUnload() method in iOS; it should be used to conduct shutdown actions such as saving persistent data.

### *b. Interface*

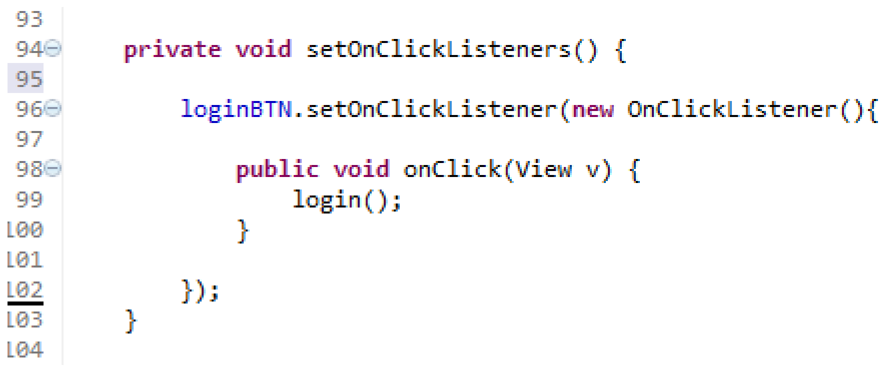
iOS and Android again differ in how their controller objects establish connections to their respective View objects. Android first loads the associated XML layout file in the onCreate() method and then establishes references to individual View objects inside the Controller object. We found it very simple to compose these functions into a single method we called setConnections(), Figure 46 shows an example setConnections() method from our LoginController.



```
    private void setConnections() {


        setContentView(R.layout.loginview);

        userTXT = (EditText) findViewById(R.id.loginViewUserTXT);
        passwordTXT = (EditText) findViewById(R.id.loginViewPasswordTXT);
        userLBL = (TextView)findViewById(R.id.loginViewUserLBL);
        loginBTN = (Button) findViewById(R.id.loginViewLoginBTN);

     }

```

Figure 46. Android setConnections Method

After initially establishing connections we needed to assign functionality to our View objects. Android Controller objects are responsible for responding to click events of View objects. In order to do this, the Controller object must establish an onClickListener which performs some action when it receives the onClick() event from the associated View object. Again we found that it was simple to establish all onClickListeners in a single method we named setOnClickListeners(). Figure 47 displays an example setOnClickListeners() function from the LoginController class. The method is very simple; the user is expected to enter required credentials and then press the login button. The LoginController waits for the onClick() event from that button and then executes the login() method.



```
    private void setOnClickListeners() {

        loginBTN.setOnClickListener(new OnClickListener(){

            public void onClick(View v) {
                login();
            }

        });
    }

```

Figure 47. Android setOnClickListener Method

When creating interfaces in iOS, the developer must create and define each object. The objects are typically created in IB and defined in the header and implementation files as an IBOutlet. After these objects have been created the developer can link the objects to the implementations in the code. In order to do this, the developer must use the drag-and-drop functionality of IB. If the ViewController has been linked to the ViewController Class files correctly, there will be a list of Outlets in the file inspector. Figure 48 shows how a developer would link the password UITextField for the NPS Muster Application.

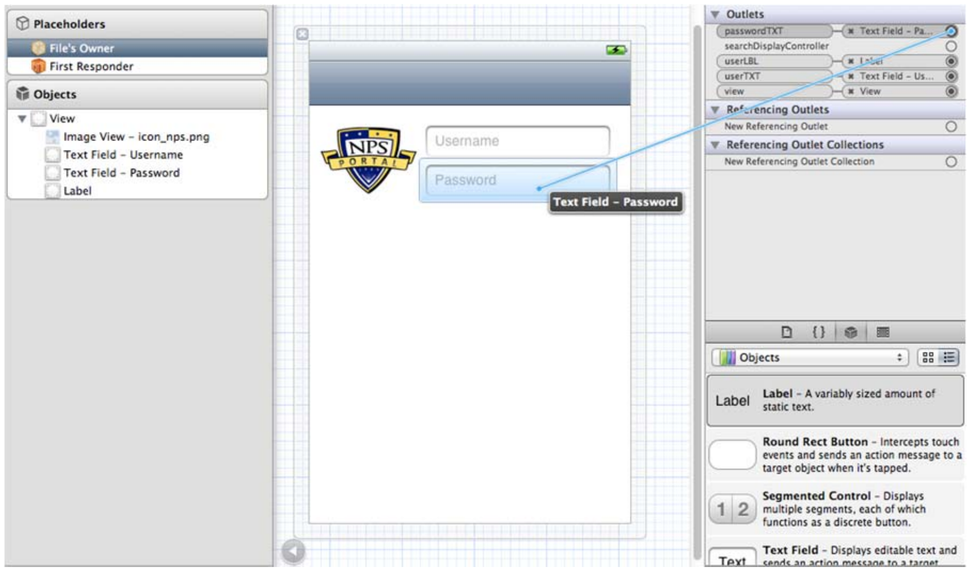

Placeholders
File's Owner
First Responder
Objects
View
Image View - icon_nps.png
Text Field - Username
Text Field - Password
Label
Username
Password
Text Field - Password
Outlets
passwordTXT
searchDisplayController
userLBL
userTXT
view
Referencing Outlets
New Referencing Outlet
Referencing Outlet Collections
New Referencing Outlet Collection
Objects
Label - A variably sized amount of static text.
Round Rect Button - Intercepts touch events and sends an action message to a target object when it's tapped.
Segmented Control - Displays multiple segments, each of which functions as a discrete button.
Text Field - Displays editable text and

Figure 48. iOS IBOutlet Link

Upon completion of this step, the ViewController has have full access to the UITextField and can manipulate it as the ViewController logic dictates.

Additionally, when a developer adds a button or some other object that links to a method, a process similar to linking objects is used to connect the object to code. An IBAction will be declared in the header file and defined in the implementation file. A similar drag-and-drop method is used to connect the method to the button. Figure 49 shows the drag-and-drop linking of the login method with the login button. After the drag-and-drop action, the developer is presented with a drop down list of events. This includes all the events that the button can handle. The most common event for a button to handle is the “Touch Up Inside” event. The “Touch Up Inside” event acts when a user presses and lifts his or her finger to the screen within the area of the UIButton.

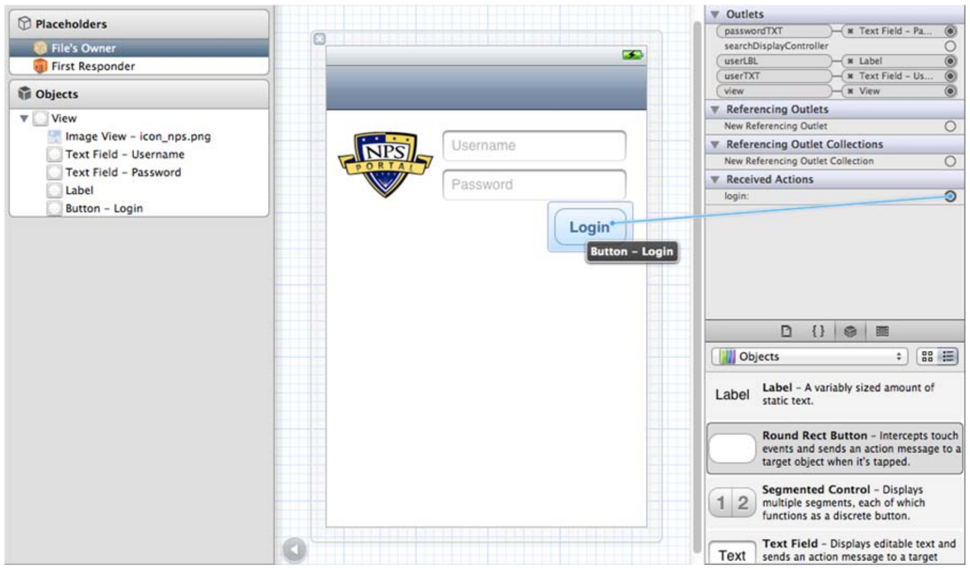

Placeholders
File's Owner
First Responder
Objects
View
Image View – icon_nps.png
Text Field – Username
Text Field – Password
Label
Button – Login
Username
Password
Login
Button – Login
Outlets
passwordTXT
Text Field – Pa...
searchDisplayController
userLBL
Label
userTXT
Text Field – Us...
view
View
Referencing Outlets
New Referencing Outlet
Referencing Outlet Collections
New Referencing Outlet Collection
Received Actions
login:
Objects
Label – A variably sized amount of static text.
Round Rect Button – Intercepts touch events and sends an action message to a target object when it's tapped.
Segmented Control – Displays multiple segments, each of which functions as a discrete button.
Text Field – Displays editable text and sends an action message to a target

Figure 49. iOS IBAction Linking

### *c. Logic*

The implementation of the underlying logic is the section that is most similar between iOS and Android Controller objects. The logic is contained in the .java (Android) and .m (iOS) files, respectively, that implement the Controller classes. If the Model objects are correctly designed it is easy to separate out the logic required to manipulate them in the Controller class.

In an independent Android application the onClickListeners may run whatever code the developer desires. iOS View objects, however, require specific target actions. By first identifying the required IBActions, it is straightforward to port those identified methods to Android. The Android onClickListener should call the same method in the onClick() callback as its iOS counterpart does in the associated View object’s IBAction.

In addition to responding to user interactions, a Controller object is also responsible for supplying the screen with required information. To this end, we developed a setView() method, which when called, should refresh the screen, ensuring that each individual View object is populated with the correct information.

It is important to note, that due to language differences, the code inside each method will be written somewhat differently between platforms. The underlying functionality should be largely the same, and many algorithms may still be reused at a more abstract level.

#### *d. Unified Controller Diagram*

The Login Controller class is implemented very differently between platforms but by identifying the functional areas we were able to simplify the design document to include only the essentials required to relay the design to a coder. It is important to note that the document, depicted in Figure 50, would prove difficult to leverage for a coder who is unfamiliar with the specific design concepts portrayed in Figure 45, the separated login controller diagrams.

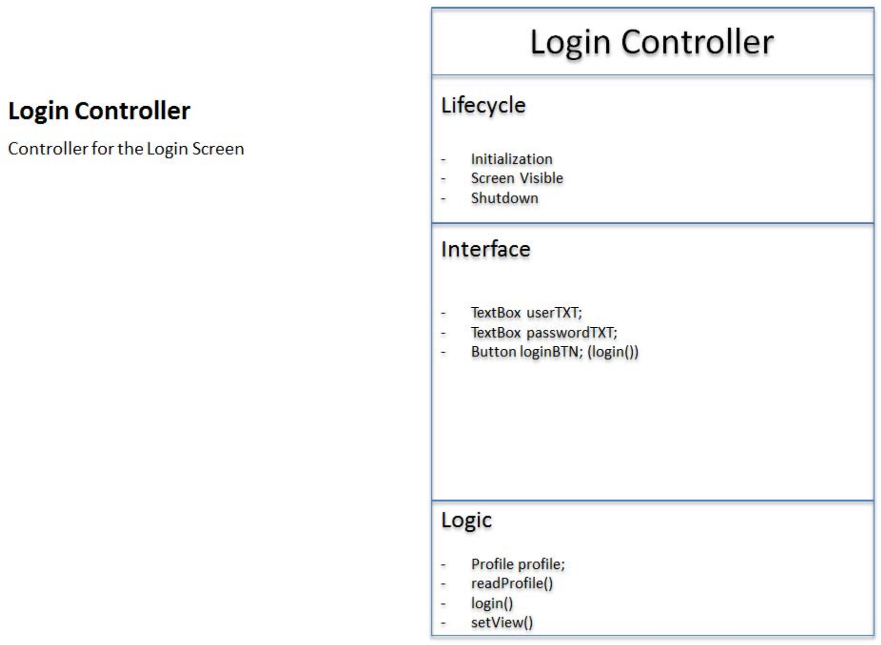


Figure 50. Login Controller Unified Diagram

## F. IMPLEMENTATION

Mobile applications are GUI intensive and focus on the user experience and interaction. We find that, using the documents described above, it is often easier to build an application utilizing a top-down approach in which the View and Controller objects are first implemented with enough functionality to exercise the required navigation hierarchy. After the proper flow of the application has been tested, required methods may be fleshed-out to provide actual functionality to the screens. The following sections will detail the process of building the NPS Muster application based on the respective design documents.

### 1. Create a New Project

Create a new Android project in Eclipse by selecting the menu items File -> New -> Android Project. On the

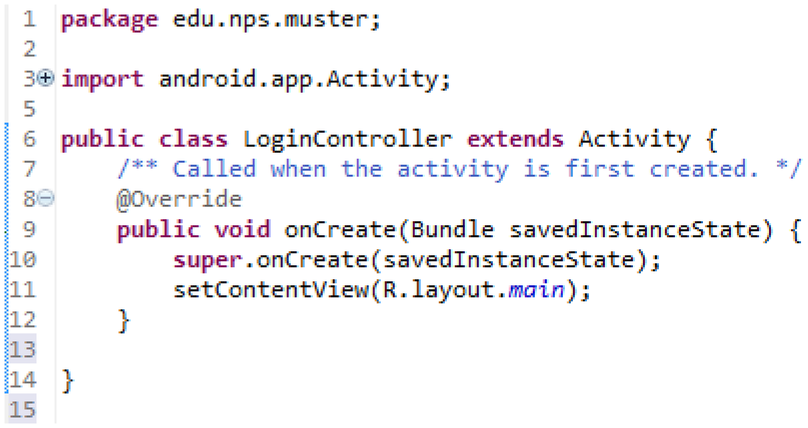
Create Android Project Screen, choose a project name [we chose **NPS Muster**] leave all other options default [we chose to work with **Android 1.6** in order to ensure maximum compatibility with older Android devices, but note that this choice will limit certain functionalities available in later versions], click next. Specify your package name in the Application Info screen, we chose edu.nps.muster. Ensure that the check box next to “Create Activity” is checked and specify the name of your main activity, for this we chose LoginController, deviating from standard Android naming conventions for the purpose of highlighting similarities between platforms. Leave all other options default and click finish, there will now be a new project, “NPS Muster,” in the Package Explorer. Two files will be created, one in the src -> edu.nps.muster folder, LoginController.java, and one in the res -> layout folder, main.xml. We recommend renaming the main.xml file to loginview.xml.

To Create a new iOS project a developer would select “New Project...” from the File -> New menu. At this point Xcode gives the developer a choice of templates to begin the project. These templates offer a starting point for the iOS project and include implementation and header files, a nib file and, often times, a rootViewController. Depending on the design documents and developer preference, these templates may be a good place to start. However, in the NPS Muster Application we started with the empty application template. This option offers the most flexibility for developer customization. After choosing a template the developer is asked to enter the desired application name, NPS Muster, and company identifier. The company identifier is similar to the Android package name. We used edu.nps.muster for the company identifier. (Note: the company identifier must match the company identifier that is registered with the provisioning profile, otherwise the application cannot be loaded to the test device) After the necessary information is entered, the project will be created by the IDE.

### 2. Add Required Controllers and Views

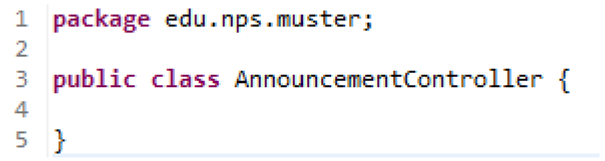
Android and iOS add screens slightly differently. Android requires that each Controller and XML layout are created individually, while iOS provides all files required to implement a screen in a simple wizard.

Android provides the main screen, including both the .java and XML layout file, with every new project; in order to add new screens to a project, one must add a new Controller and a new XML layout. To add a new Controller, right click the src -> edu.nps.muster folder and select New -> Class. Change the name to correspond with a Controller in the class diagrams and click “finish.” Figure 51 shows the default Activity Android provides with a new project, Figure 52 shows a new class before being tailored by the developer.



```
package edu.nps.muster;

import android.app.Activity;

public class LoginController extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

}
```

Figure 51. Default Android Activity

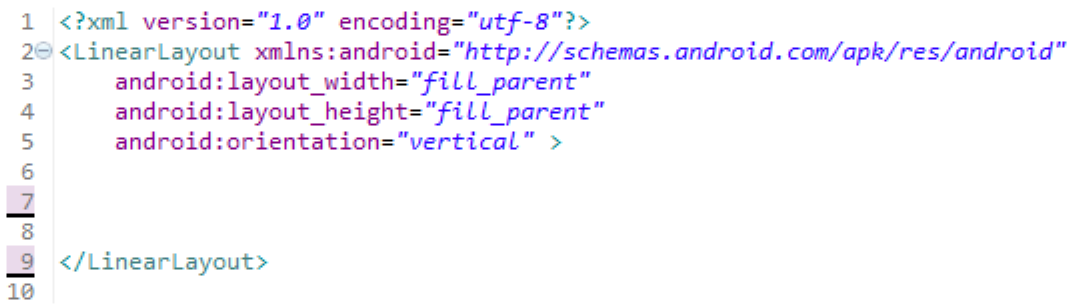


```
package edu.nps.muster;

public class AnnouncementController {

}
```

Figure 52. Default Android Class

View layouts may be created in Eclipse through the Graphical Layout Editor. This editor provides drag-and-drop functionality by which different View objects may be arranged visually on the screen. Additionally, it provides direct access to the XML document, allowing developers to modify elements in a very precise manner.

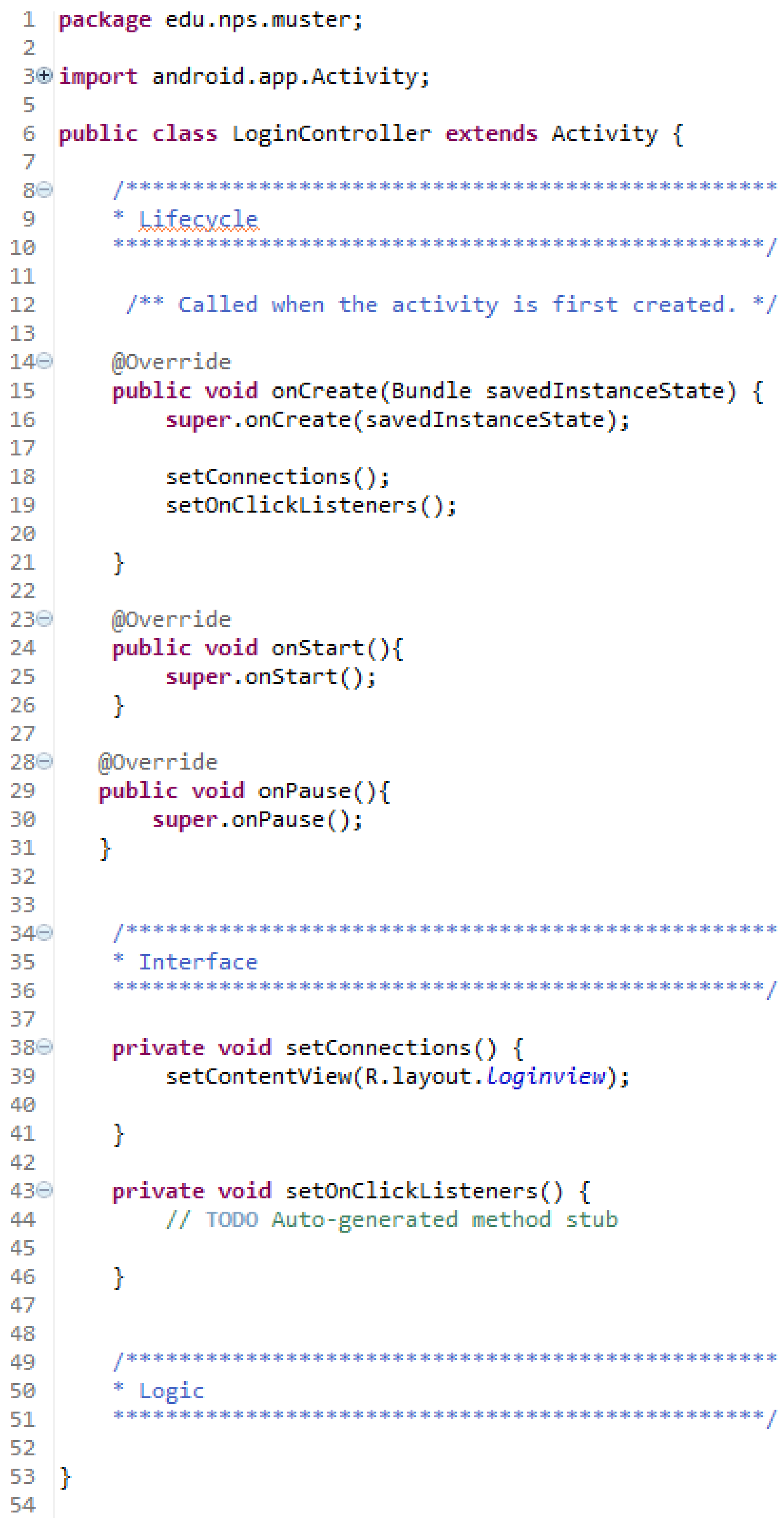
XML layout documents for Android projects are stored in the res -> layout folder. To create a new document, right click on the res -> layout folder, select New -> Other -> Android -> Android XML Layout File. Choose a name that corresponds to a controller with which the new layout should be associated. New View layouts are created with a simple LinearLayout to which other Views will be added. Figure 53 shows the newly created loginview.xml.



```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >



</LinearLayout>
```

Figure 53. Default View XML

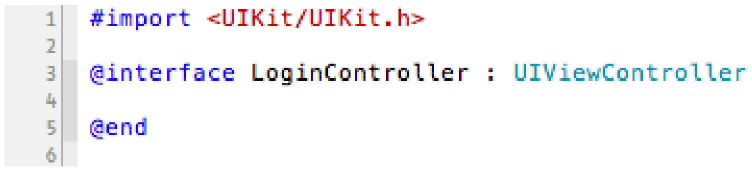
Each controller should be modified in order to align with its associated class diagram. Open the .java file by double clicking the file name in the project tree. Add the setConnections() and setOnClickListener() methods and call them in the onCreate() lifecycle callback. Move the setContentView(R.layout.loginview) call into the setConnections() method. Repeat this process for each screen in the application. Figure 54 shows an Android controller object after it has been modified according to using these steps. We recommend adding comments to delineate the functional areas of the controller.



```java
package edu.nps.muster;

import android.app.Activity;

public class LoginController extends Activity {

    /*************************************************
    * Lifecycle
    *************************************************/

     /** Called when the activity is first created. */

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setConnections();
        setOnClickListeners();

    }

    @Override
    public void onStart(){
        super.onStart();
    }

   @Override
   public void onPause(){
       super.onPause();
   }


    /*************************************************
    * Interface
    *************************************************/

    private void setConnections() {
        setContentView(R.layout.loginview);

    }

    private void setOnClickListeners() {
        // TODO Auto-generated method stub

    }


    /*************************************************
    * Logic
    *************************************************/

}
```

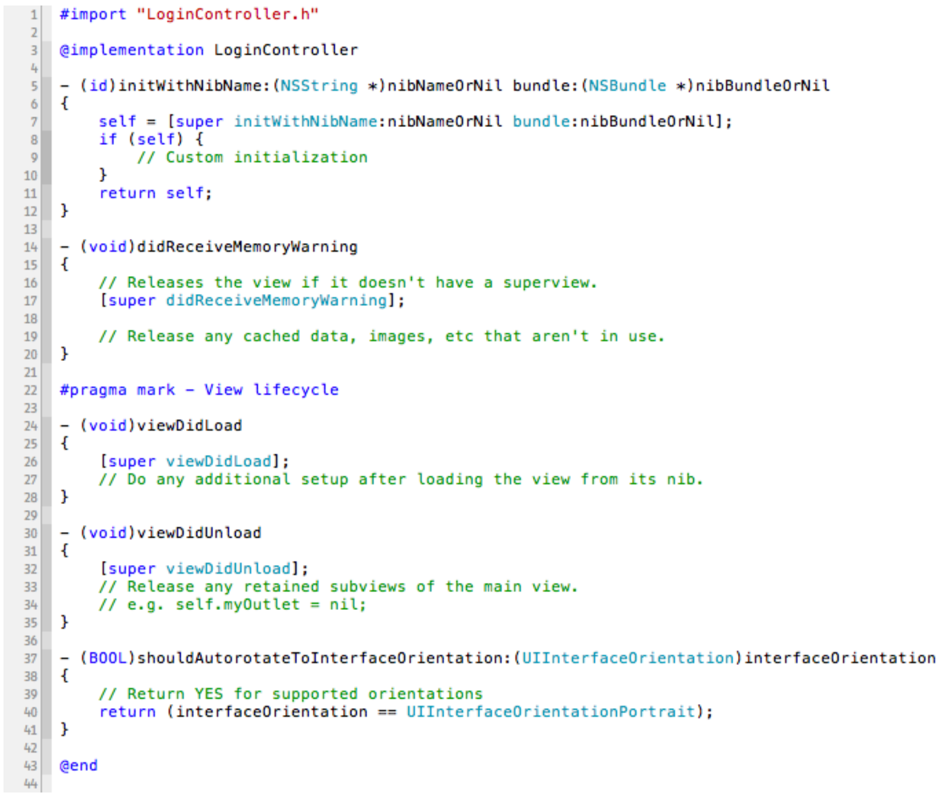
Figure 54.    Android Functional Areas

In iOS, when using the empty application template, the developer is given only the appDelegate header and implementation files. The first step in adding a new screen is to choose the “New File” option from the File -> New menu. This will give the developer the options as to what type of file to add. Since we are trying to add a new view and viewController, choose the UIViewController Subclass option. In the next option window ensure that the selected subclass is UIViewController, not UITableViewController, and that the “With xib for User Intereface” option is checked. Give the file a name and choose create. This process should be repeated for each of the screens needed for the project. For screens that require a table or list of options, like the announcementList, choose to subclass as a UITableViewController in the second setup window. Figures 55 and 56 show the default header and implementation files, respectively, provided by Xcode.



```
#import <UIKit/UIKit.h>

@interface LoginController : UIViewController

@end

```

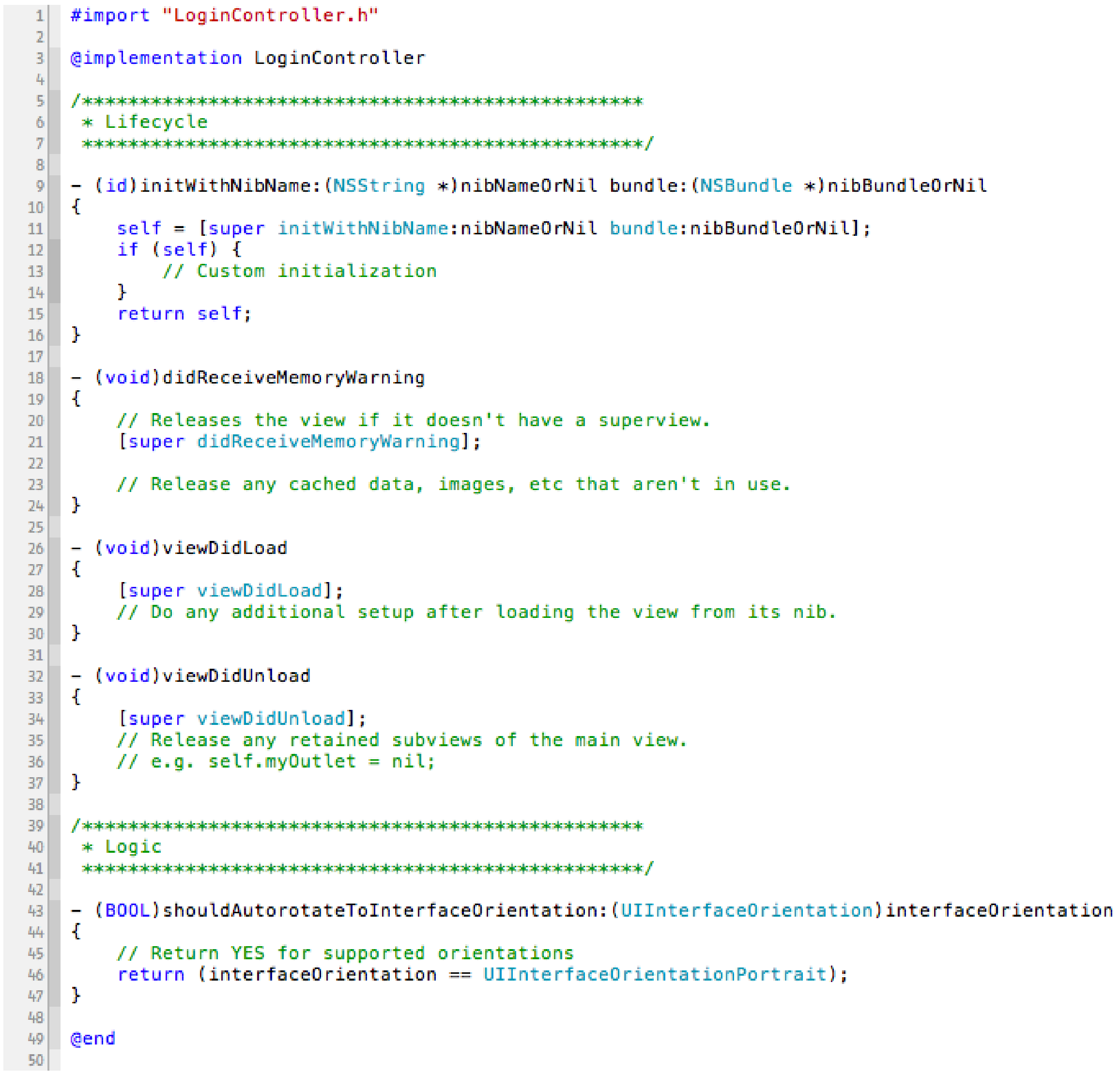
Figure 55. Default iOS Header File



```
#import "LoginController.h"

@implementation LoginController

- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
        // Custom initialization
    }
    return self;
}

- (void)didReceiveMemoryWarning
{
    // Releases the view if it doesn't have a superview.
    [super didReceiveMemoryWarning];

    // Release any cached data, images, etc that aren't in use.
}

#pragma mark - View lifecycle

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view from its nib.
}

- (void)viewDidUnload
{
    [super viewDidUnload];
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
}

- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation
{
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

@end
```

Figure 56.    Default iOS Implementation File

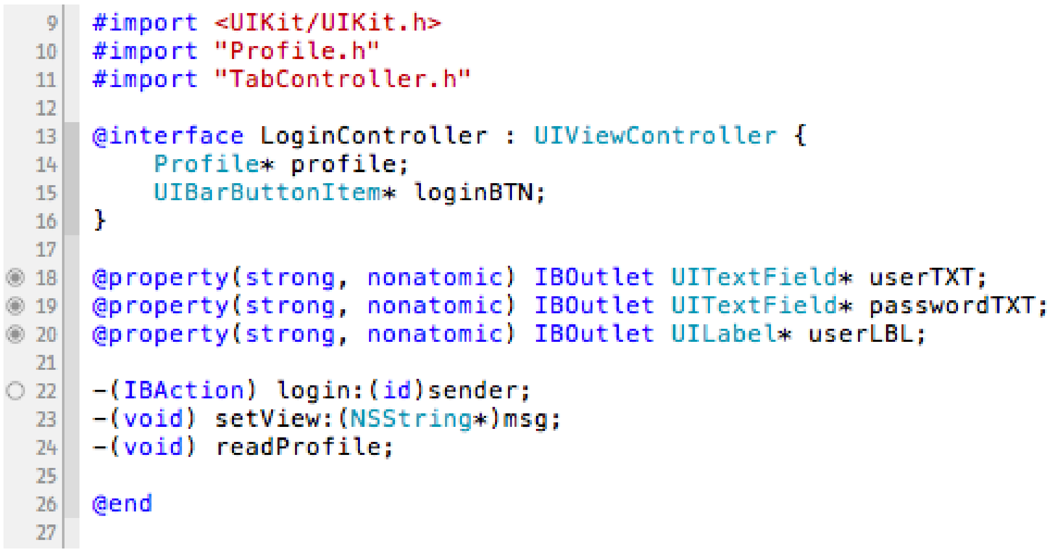
When we modify the iOS controller code to show the lifecycle, logic, and interface functional areas, we found that the header file contained the interface and the implementation file contained the logic and lifecycle methods, which can be broken up as shown in Figure 57.



```
#import "LoginController.h"

@implementation LoginController

/****************************************************
 * Lifecycle
 ****************************************************/

- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
        // Custom initialization
    }
    return self;
}

- (void)didReceiveMemoryWarning
{
    // Releases the view if it doesn't have a superview.
    [super didReceiveMemoryWarning];

    // Release any cached data, images, etc that aren't in use.
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view from its nib.
}

- (void)viewDidUnload
{
    [super viewDidUnload];
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
}

/****************************************************
 * Logic
 ****************************************************/

- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation
{
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

@end
```

Figure 57. iOS Functional Areas

At this point the developer can begin to build the interface functional area by defining IBOutlets and IBActions in the header file. The IBOutlets and IBActions should be added to align with the objects and methods defined in the unified controller diagram. The IBActions do not need to be fully defined in the implementation file; however, it is good to use stub-code as placeholders. Once all the necessary interface objects and methods are defined the developer can begin linking objects and functions, as previously shown. The header file, which contains the interface functional area for the iOS login screen, is shown in Figure 58.

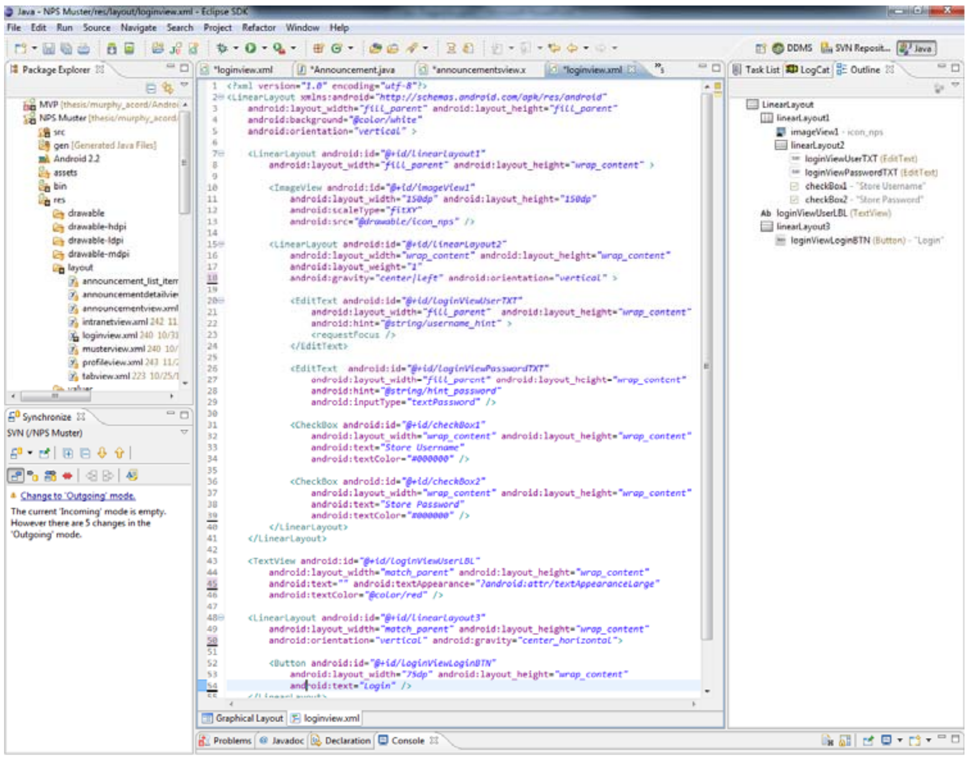


```
#import <UIKit/UIKit.h>
#import "Profile.h"
#import "TabController.h"

@interface LoginController : UIViewController {
    Profile* profile;
    UIBarButtonItem* loginBTN;
}

@property(strong, nonatomic) IBOutlet UITextField* userTXT;
@property(strong, nonatomic) IBOutlet UITextField* passwordTXT;
@property(strong, nonatomic) IBOutlet UILabel* userLBL;

-(IBAction) login:(id)sender;
-(void) setView:(NSString*)msg;
-(void) readProfile;

@end
```

Figure 58. iOS LoginController header file

### 3. Edit Required Views

Each screen will require a View hierarchy associated with a Controller object. As Views align with platform specifications and appearance, developers may expect individual Views to differ between platforms. Attempts should be made to maintain a common “feel” while fully embracing the individual platform user experience, that is, the target platform “look-and-feel.” Figures 59 and 60 show the finished loginview.xml, in XML and graphically, respectively.



```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent" android:layout_height="fill_parent"
    android:background="@color/white"
    android:orientation="vertical" >

    <LinearLayout android:id="@+id/linearLayout1"
        android:layout_width="fill_parent" android:layout_height="wrap_content" >

        <ImageView android:id="@+id/imageView1"
            android:layout_width="150dp" android:layout_height="150dp"
            android:scaleType="fitXY"
            android:src="@drawable/icon_nps" />

        <LinearLayout android:id="@+id/linearLayout2"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:layout_weight="1"
            android:gravity="center|left" android:orientation="vertical" >

            <EditText android:id="@+id/loginViewUserTXT"
                android:layout_width="fill_parent"  android:layout_height="wrap_content"
                android:hint="@string/username_hint" >
                <requestFocus />
            </EditText>

            <EditText  android:id="@+id/loginViewPasswordTXT"
                android:layout_width="fill_parent" android:layout_height="wrap_content"
                android:hint="@string/hint_password"
                android:inputType="textPassword" />

            <CheckBox android:id="@+id/checkBox1"
                android:layout_width="wrap_content" android:layout_height="wrap_content"
                android:text="Store Username"
                android:textColor="#000000" />

            <CheckBox android:id="@+id/checkBox2"
                android:layout_width="wrap_content" android:layout_height="wrap_content"
                android:text="Store Password"
                android:textColor="#000000" />
        </LinearLayout>
    </LinearLayout>

    <TextView android:id="@+id/loginViewUserLBL"
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:text="" android:textAppearance="?android:attr/textAppearanceLarge"
        android:textColor="@color/red" />

    <LinearLayout android:id="@+id/linearLayout3"
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:orientation="vertical" android:gravity="center_horizontal">

        <Button android:id="@+id/loginViewLoginBTN"
            android:layout_width="75dp" android:layout_height="wrap_content"
            android:text="Login" />
```

Figure 59. Finished View Code

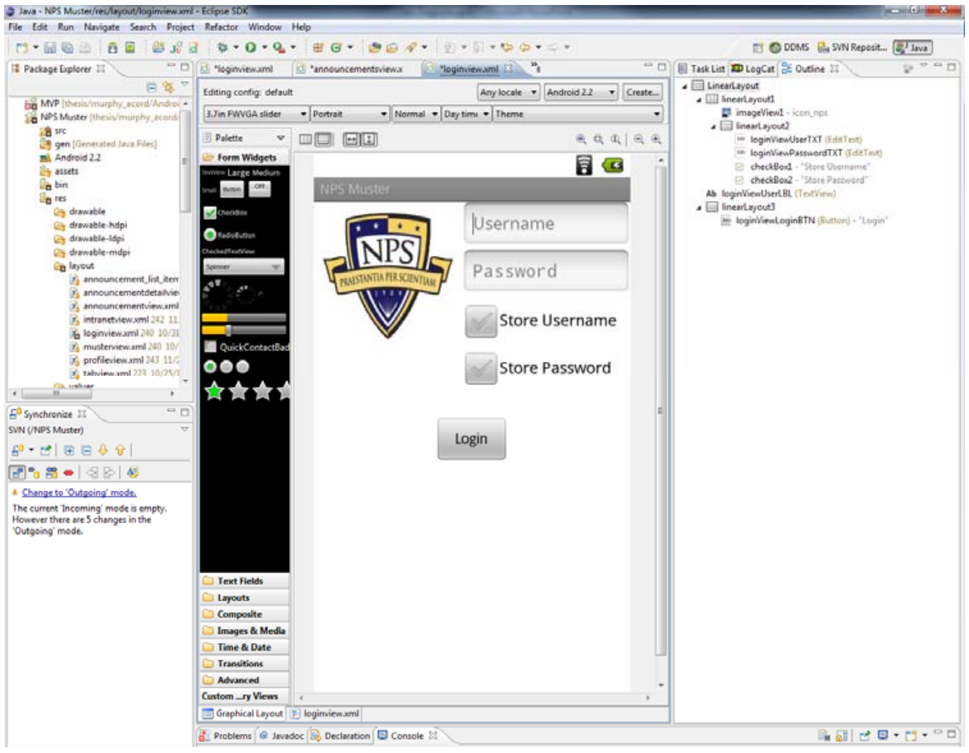


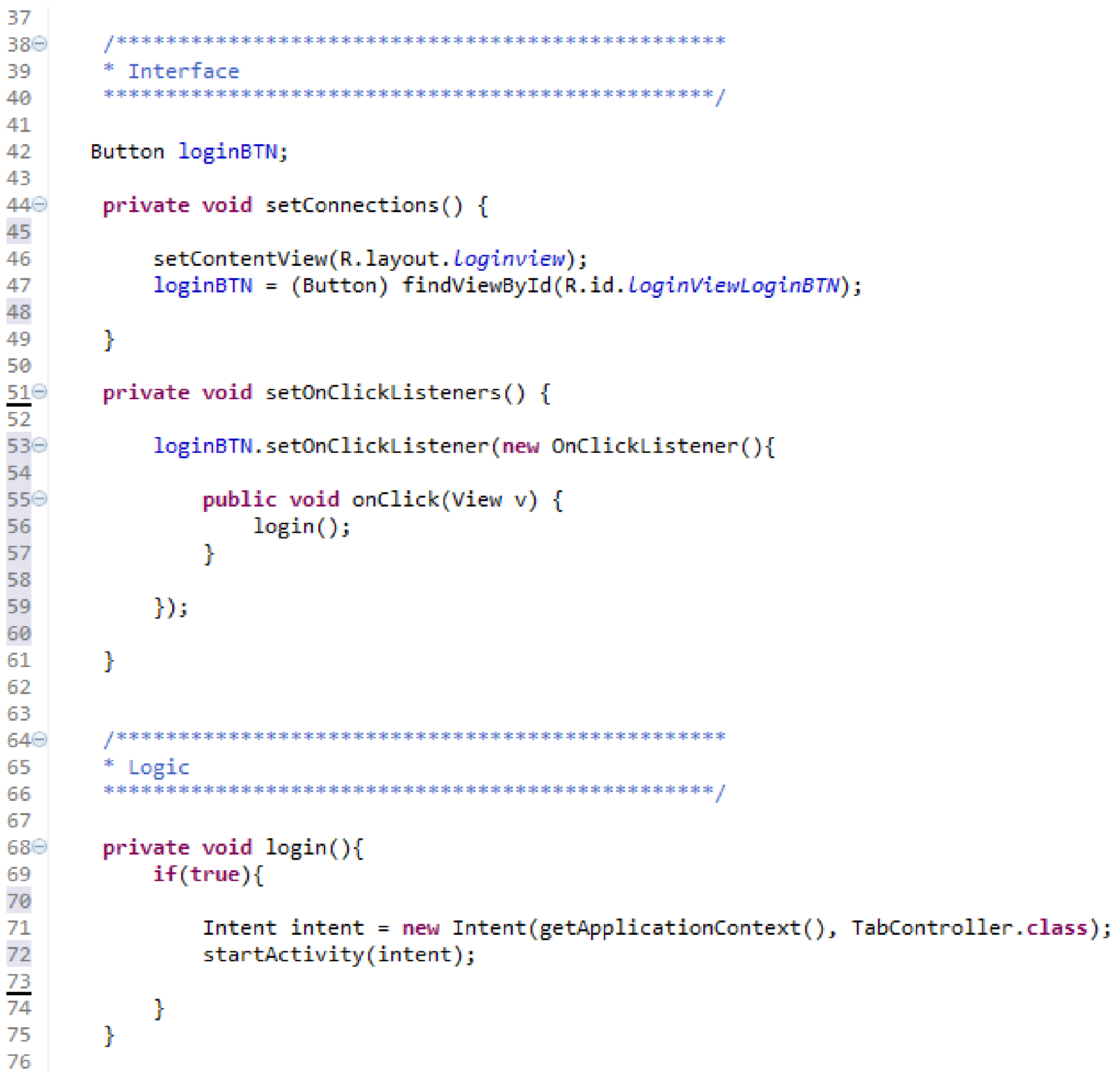
Figure 60. Finished View Graphical

Editing the nib files is all done using the IB. IB provides the developer a graphical view of the display and a list of objects that can be added to the screen. The developer then selects and edits each object’s placement, size, and visual options. This step must be completed before the developer can link IBOutlets and IBActions, previously shown in Figures 48 and 49.

### 4. Implement Navigation

The Login screen will be required to push another screen on top of it upon the user’s successful login to the network. The LoginController class diagram calls for a login() method which will use the Profile model to verify credentials and, if successful, allow the user to proceed to the next screen. We had not implemented the Profile class at this point in the development, so the proper functionality was assumed in this step.

To implement the navigation portion of this process in Android, reference the login button from the XML layout in the setConnections() method, add the login() method in the logic section of the LoginController, and call it in the onClickListener of the Login button. At this point the logic and interface sections in LoginController.java should look like Figure 61.

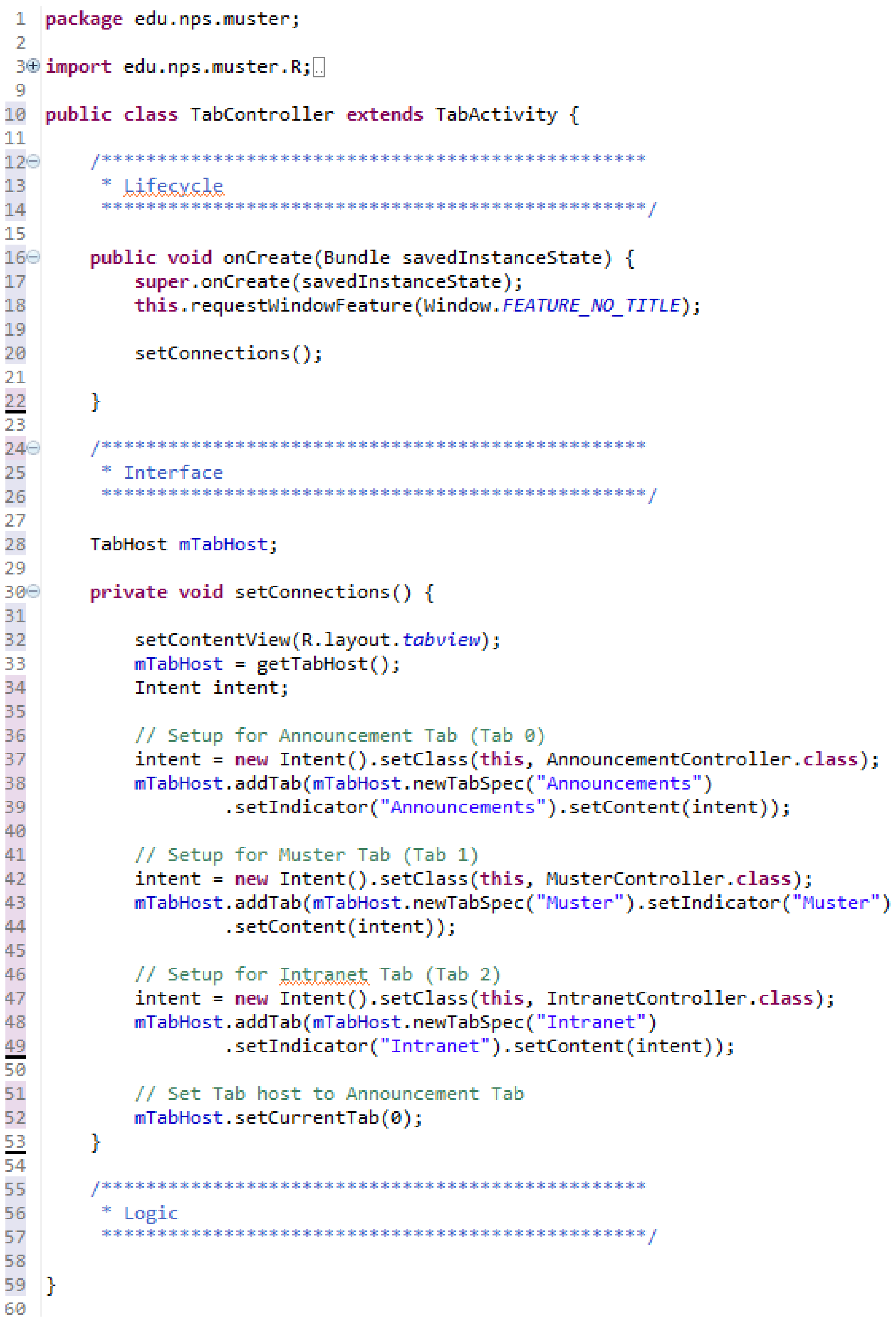


```

    /************************************************
     * Interface
     ************************************************/

   Button loginBTN;

    private void setConnections() {

        setContentView(R.layout.loginview);
        loginBTN = (Button) findViewById(R.id.loginViewLoginBTN);

    }

    private void setOnClickListeners() {

        loginBTN.setOnClickListener(new OnClickListener(){

            public void onClick(View v) {
                login();
            }

        });

    }


    /************************************************
     * Logic
     ************************************************/

    private void login(){
        if(true){

            Intent intent = new Intent(getApplicationContext(), TabController.class);
            startActivity(intent);

        }
    }

```

Figure 61.    LoginController Code

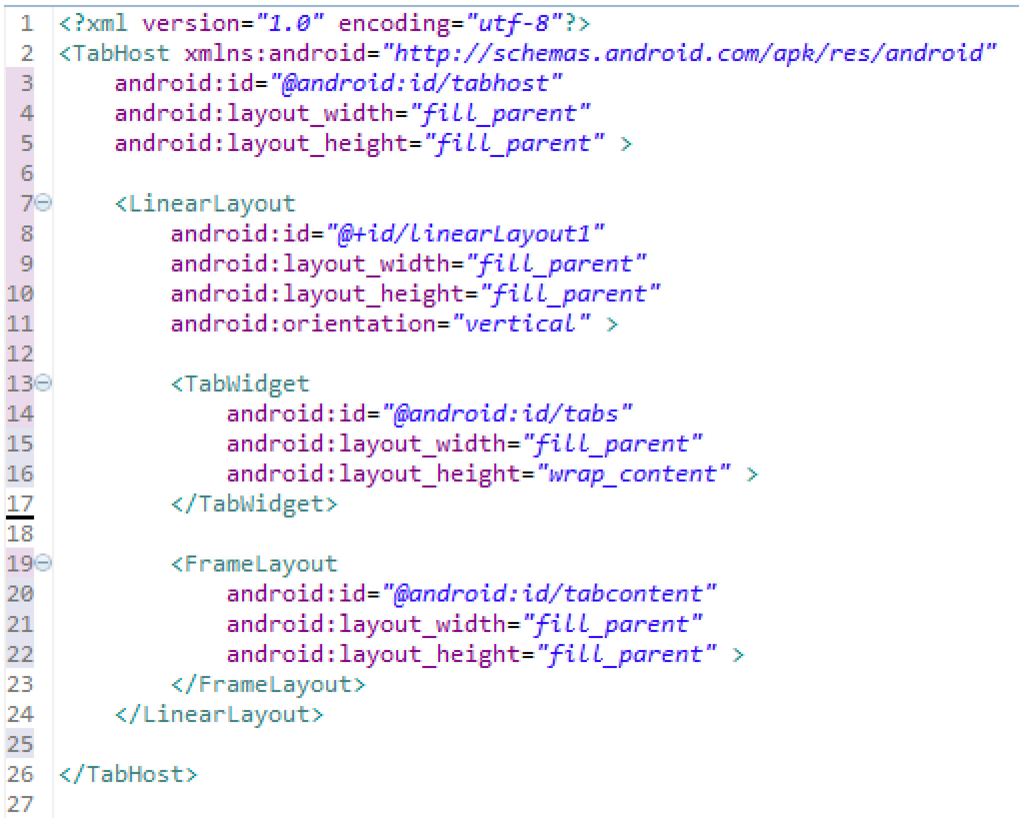
The screen pushed on top of the Login screen is part of a horizontal navigation structure. We must ensure that the horizontal navigation is implemented correctly and that all child screens are also properly represented. Again, at this point actual functionality of individual screens is not required; we are focusing simply on ensuring proper navigation-flow throughout the application. We used pseudo-functions that simulated the data received from the models to ensure that the navigation could be tested, without first having to develop the models.

As stated previously, Android implements a subclass of TabActivity to implement any required tabs. We chose to name our subclass TabController. In order to create this controller, follow the procedure outlined above for creating a new class and modify it to align with Figure 62. TabActivities require a specific view layout, which includes a TabHost that will be modified through code. Use Figure 63 as a template when creating view layouts associated withTabActivities.



```java
package edu.nps.muster;

import edu.nps.muster.R;

public class TabController extends TabActivity {

    /**************************************************
     * Lifecycle
     **************************************************/

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        this.requestWindowFeature(Window.FEATURE_NO_TITLE);

        setConnections();

    }

    /**************************************************
     * Interface
     **************************************************/

    TabHost mTabHost;

    private void setConnections() {

        setContentView(R.layout.tabview);
        mTabHost = getTabHost();
        Intent intent;

        // Setup for Announcement Tab (Tab 0)
        intent = new Intent().setClass(this, AnnouncementController.class);
        mTabHost.addTab(mTabHost.newTabSpec("Announcements")
                .setIndicator("Announcements").setContent(intent));

        // Setup for Muster Tab (Tab 1)
        intent = new Intent().setClass(this, MusterController.class);
        mTabHost.addTab(mTabHost.newTabSpec("Muster").setIndicator("Muster")
                .setContent(intent));

        // Setup for Intranet Tab (Tab 2)
        intent = new Intent().setClass(this, IntranetController.class);
        mTabHost.addTab(mTabHost.newTabSpec("Intranet")
                .setIndicator("Intranet").setContent(intent));

        // Set Tab host to Announcement Tab
        mTabHost.setCurrentTab(0);
    }

    /**************************************************
     * Logic
     **************************************************/

}
```

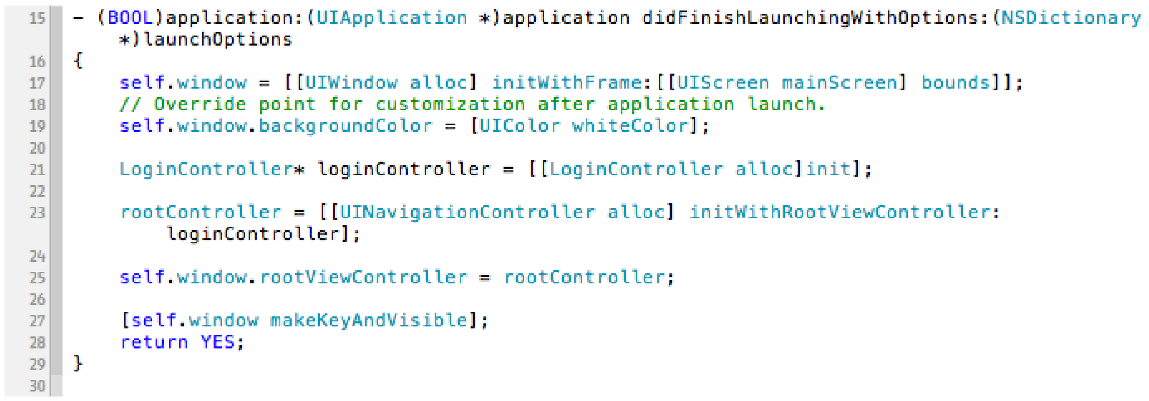
Figure 62.    TabController Code



```xml
<?xml version="1.0" encoding="utf-8"?>
<TabHost xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/tabhost"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >

    <LinearLayout
        android:id="@+id/linearLayout1"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:orientation="vertical" >

        <TabWidget
            android:id="@android:id/tabs"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content" >
        </TabWidget>

        <FrameLayout
            android:id="@android:id/tabcontent"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent" >
        </FrameLayout>
    </LinearLayout>

</TabHost>
```

Figure 63.    TabController View Code

The first step in implementing navigation on iOS is to edit the appDelegate. In the application:didFinishLaunchingWithOptions:launchOptions function, which is show in Figure 64, we have instantiated a LoginController and set it as the rootController for the application. This step will set the LoginController as the first screen to be shown upon application startup.

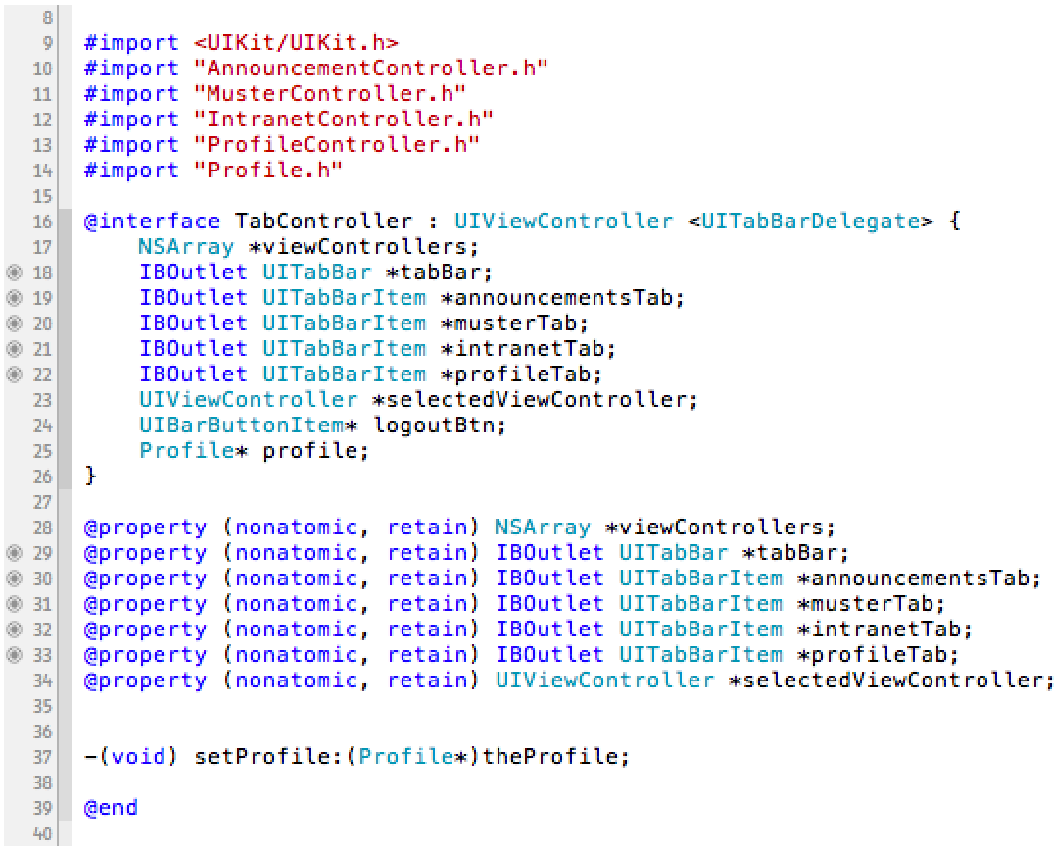


```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary
    *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    self.window.backgroundColor = [UIColor whiteColor];

    LoginController* loginController = [[LoginController alloc]init];

    rootController = [[UINavigationController alloc] initWithRootViewController:
        loginController];

    self.window.rootViewController = rootController;

    [self.window makeKeyAndVisible];
    return YES;
}
```

Figure 64.    NPS Muster Application AppDelegate

If the entire navigation scheme of the application is stack based, the rootController will be the only controller needed. Likewise, if the navigation scheme is solely horizontal, only a tabViewController will need to be implemented. However, in the NPS Muster application we use a composite navigation scheme. Our design process required a complicated combination of stack and horizontal navigation. While this implementation is not preferred by the Apple UI Guidelines, we chose this layout because it followed our UI Navigation Diagram, Figure 41.

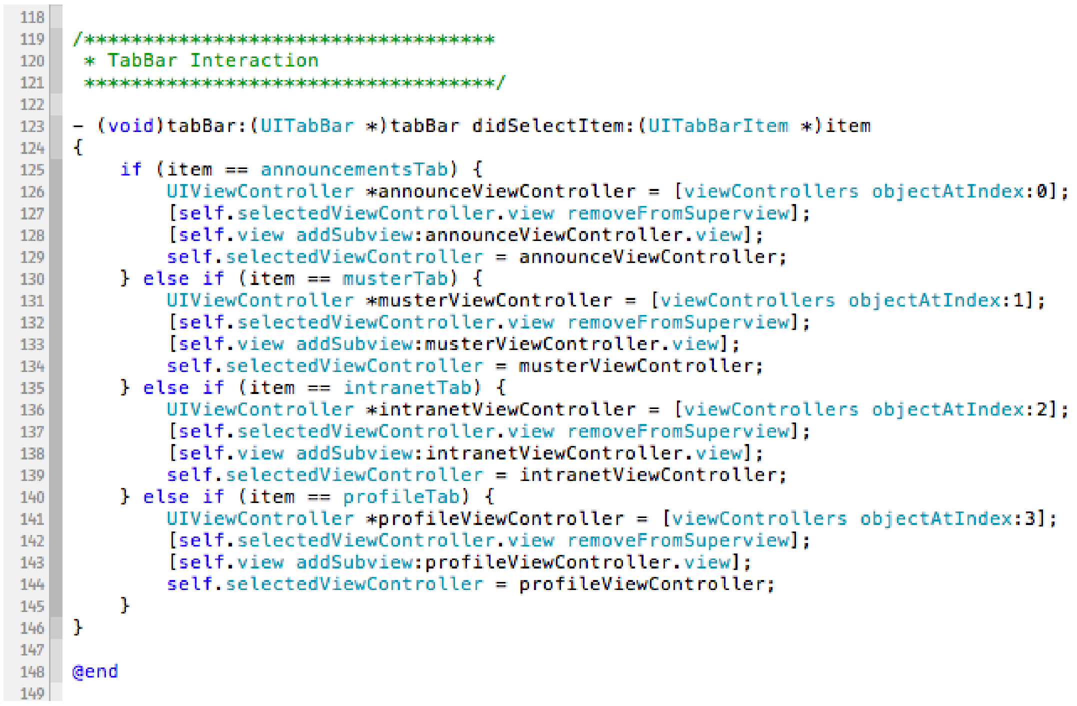
In order to accomplish our composite navigation, we had to write our own subclass of tabViewController. Creating a new UIViewController and defining it as a UITabBarDelegate can accomplish this. Figure 65 shows the header file for our tabController. Line 16 shows how we defined the tabController as a UITabBarDelegate.



```

#import <UIKit/UIKit.h>
#import "AnnouncementController.h"
#import "MusterController.h"
#import "IntranetController.h"
#import "ProfileController.h"
#import "Profile.h"

@interface TabController : UIViewController <UITabBarDelegate> {
    NSArray *viewControllers;
    IBOutlet UITabBar *tabBar;
    IBOutlet UITabBarItem *announcementsTab;
    IBOutlet UITabBarItem *musterTab;
    IBOutlet UITabBarItem *intranetTab;
    IBOutlet UITabBarItem *profileTab;
    UIViewController *selectedViewController;
    UIBarButtonItem* logoutBtn;
    Profile* profile;
}

@property (nonatomic, retain) NSArray *viewControllers;
@property (nonatomic, retain) IBOutlet UITabBar *tabBar;
@property (nonatomic, retain) IBOutlet UITabBarItem *announcementsTab;
@property (nonatomic, retain) IBOutlet UITabBarItem *musterTab;
@property (nonatomic, retain) IBOutlet UITabBarItem *intranetTab;
@property (nonatomic, retain) IBOutlet UITabBarItem *profileTab;
@property (nonatomic, retain) UIViewController *selectedViewController;


-(void) setProfile:(Profile*)theProfile;

@end

```

Figure 65.    iOS TabController Header File

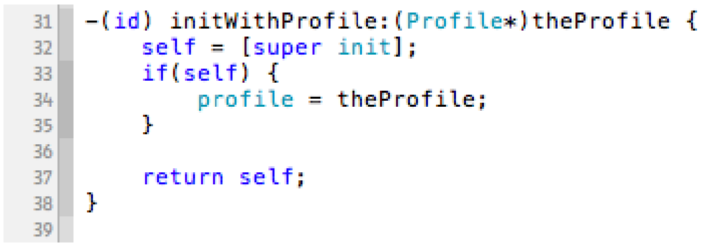
After we defined the UITabBarDelegate, we implemented the tab interaction logic, shown in Figure 66. The tab interaction logic handles the touch events that correspond to screen selection, as shown.



```
/***********************************
 * TabBar Interaction
 ***********************************/

- (void)tabBar:(UITabBar *)tabBar didSelectItem:(UITabBarItem *)item
{
    if (item == announcementsTab) {
        UIViewController *announceViewController = [viewControllers objectAtIndex:0];
        [self.selectedViewController.view removeFromSuperview];
        [self.view addSubview:announceViewController.view];
        self.selectedViewController = announceViewController;
    } else if (item == musterTab) {
        UIViewController *musterViewController = [viewControllers objectAtIndex:1];
        [self.selectedViewController.view removeFromSuperview];
        [self.view addSubview:musterViewController.view];
        self.selectedViewController = musterViewController;
    } else if (item == intranetTab) {
        UIViewController *intranetViewController = [viewControllers objectAtIndex:2];
        [self.selectedViewController.view removeFromSuperview];
        [self.view addSubview:intranetViewController.view];
        self.selectedViewController = intranetViewController;
    } else if (item == profileTab) {
        UIViewController *profileViewController = [viewControllers objectAtIndex:3];
        [self.selectedViewController.view removeFromSuperview];
        [self.view addSubview:profileViewController.view];
        self.selectedViewController = profileViewController;
    }
}

@end
```

Figure 66. iOS TabBar Interaction Code

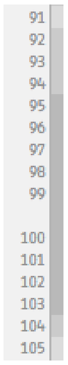
We also created our own initialization method so that the loginController could pass the profile to the tabController, as shown in Figure 67. This step allowed our application to have access to the profile regardless of the tab selected. This is not the only way to implement this functionality, but is the way we chose to implement the profile passing. We chose this method of passing the model because it more closely aligned with the Android getExtra() implementation.



```
-(id) initWithProfile:(Profile*)theProfile {
    self = [super init];
    if(self) {
        profile = theProfile;
    }

    return self;
}
```

Figure 67. iOS initWithProfile

At this point, we needed to stub the loginControllers login function so that we could implement the transition between the rootController and the tabController. Figure 68 shows the stub-code to accomplish the transition.



```
-(void)login:(id)sender {

    if(true) {

        TabController* tabController = [[TabController alloc] init];

        self.navigationItem.backBarButtonItem = [[UIBarButtonItem alloc] initWithTitle:
            @"Logout" style:UIBarButtonItemStylePlain target:nil action:nil];

        [self.navigationController pushViewController:tabController animated:YES];

    }
}
```

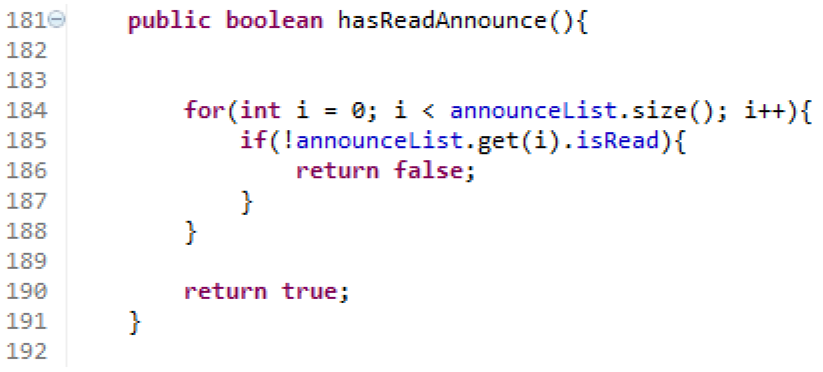
Figure 68. iOS Stubbed Login Method

### 5. Implement Models

Implementation of Model classes is straight forward on either platform, and requires almost no deviation from the design documents. When coding, developers can expect to experience minor language differences that may require slight modifications, but the overall structure of the class should remain intact. Also, with the difference in APIs, the developer may encounter other significant issues. For instance, the timer class in Android is based on milliseconds, while the iOS timer is based on seconds. Failure to account for such differences can result in significant application functional discrepancies, and even application failure.

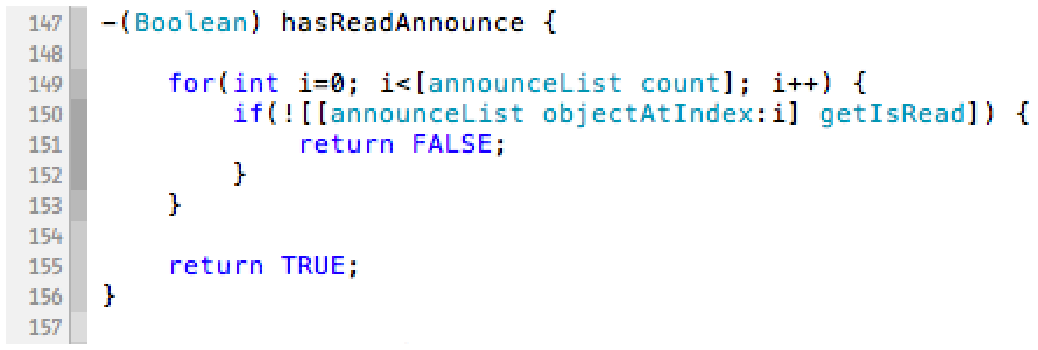
Below are several examples of functions we have implemented in our Profile and Announcement classes. These functions provide the majority of functionality and all of the data storage requirements for the application. This offloads the requirement from the Controller objects, allowing Controllers to be focused solely on controlling View and Model objects.

Figure 69 and 70 are the hasReadAnnounce() methods in the Profile class. Is scans through the Profile’s array of announcements and checks for unread announcements. These code-snippets also serve to highlight the similarity and differences in the languages used for each platform. The underlying logic is consistent.



```
    public boolean hasReadAnnounce(){


        for(int i = 0; i < announceList.size(); i++){
            if(!announceList.get(i).isRead){
                return false;
            }
        }

        return true;
    }

```

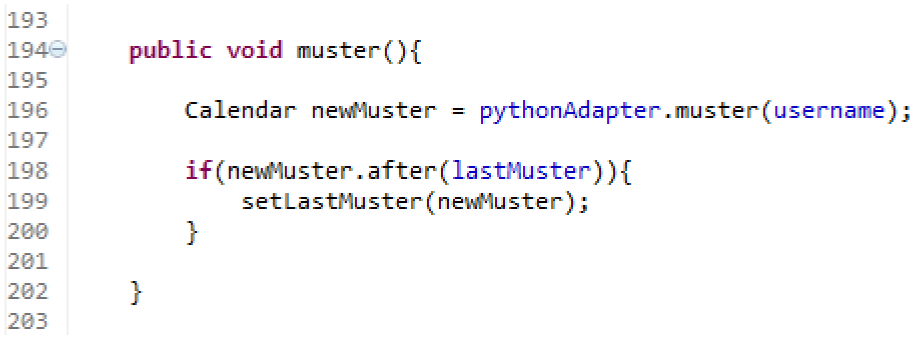
Figure 69. hasReadAnnounce() Android Method



```
-(Boolean) hasReadAnnounce {

    for(int i=0; i<[announceList count]; i++) {
        if(![[announceList objectAtIndex:i] getIsRead]) {
            return FALSE;
        }
    }

    return TRUE;
}

```

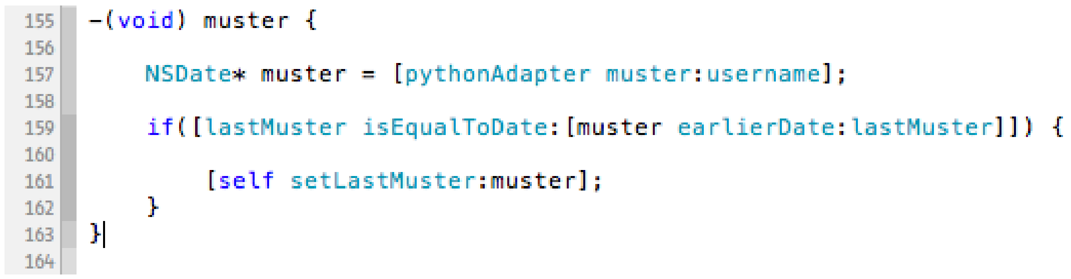
Figure 70. hasReadAnnounce iOS Method

Figure 71 and 72 are the muster() methods. It utilizes the PythonDBAdapter class to conduct a muster with the remote database. A successful muster attempt returns current date and time, an unsuccessful muster returns the last muster date and time.



```
    public void muster(){

        Calendar newMuster = pythonAdapter.muster(username);

        if(newMuster.after(lastMuster)){
            setLastMuster(newMuster);
        }

    }
```

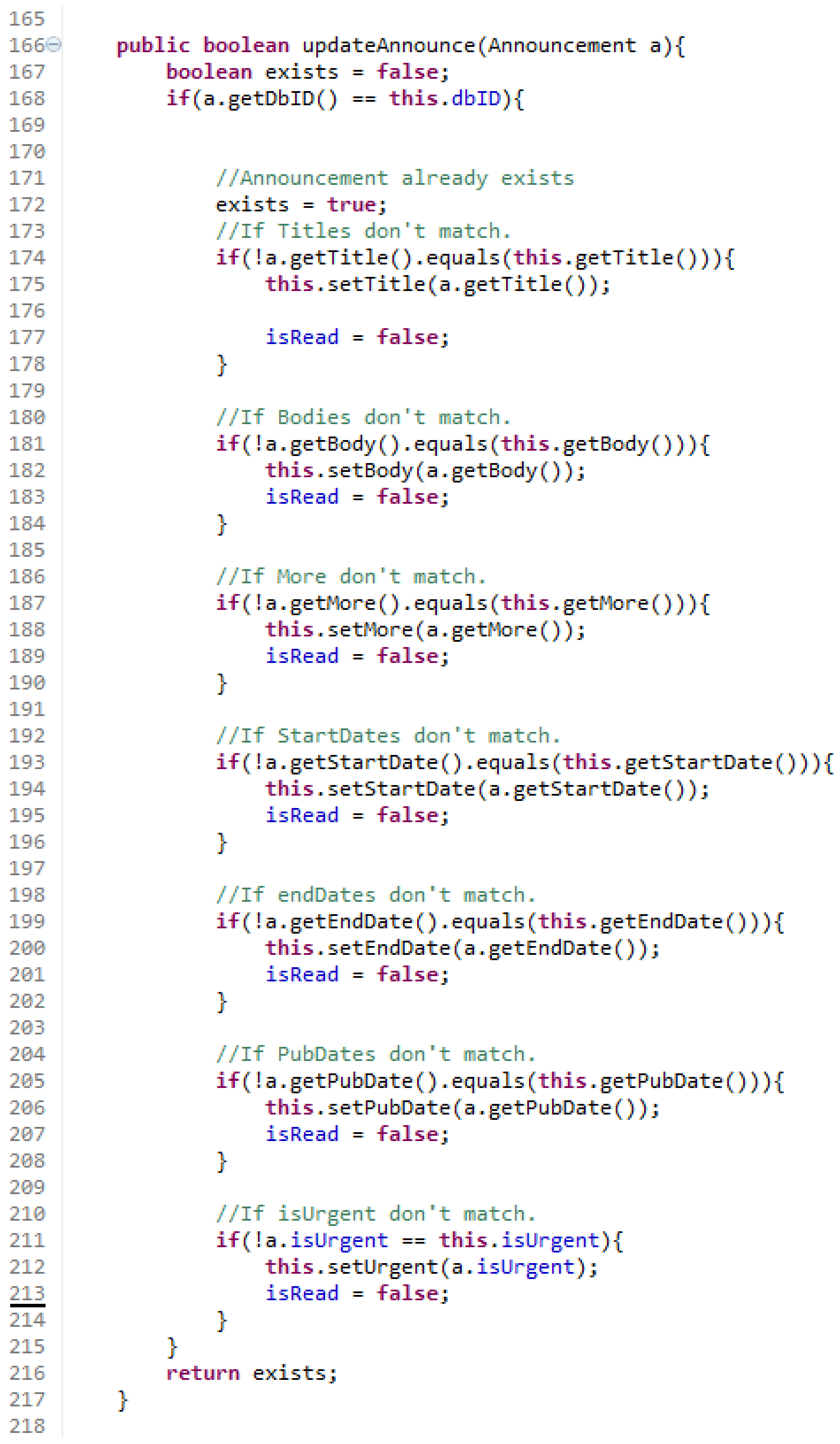
Figure 71. muster() Android Method



```
-(void) muster {

    NSDate* muster = [pythonAdapter muster:username];

    if([lastMuster isEqualToDate:[muster earlierDate:lastMuster]]) {

        [self setLastMuster:muster];
    }
}
```

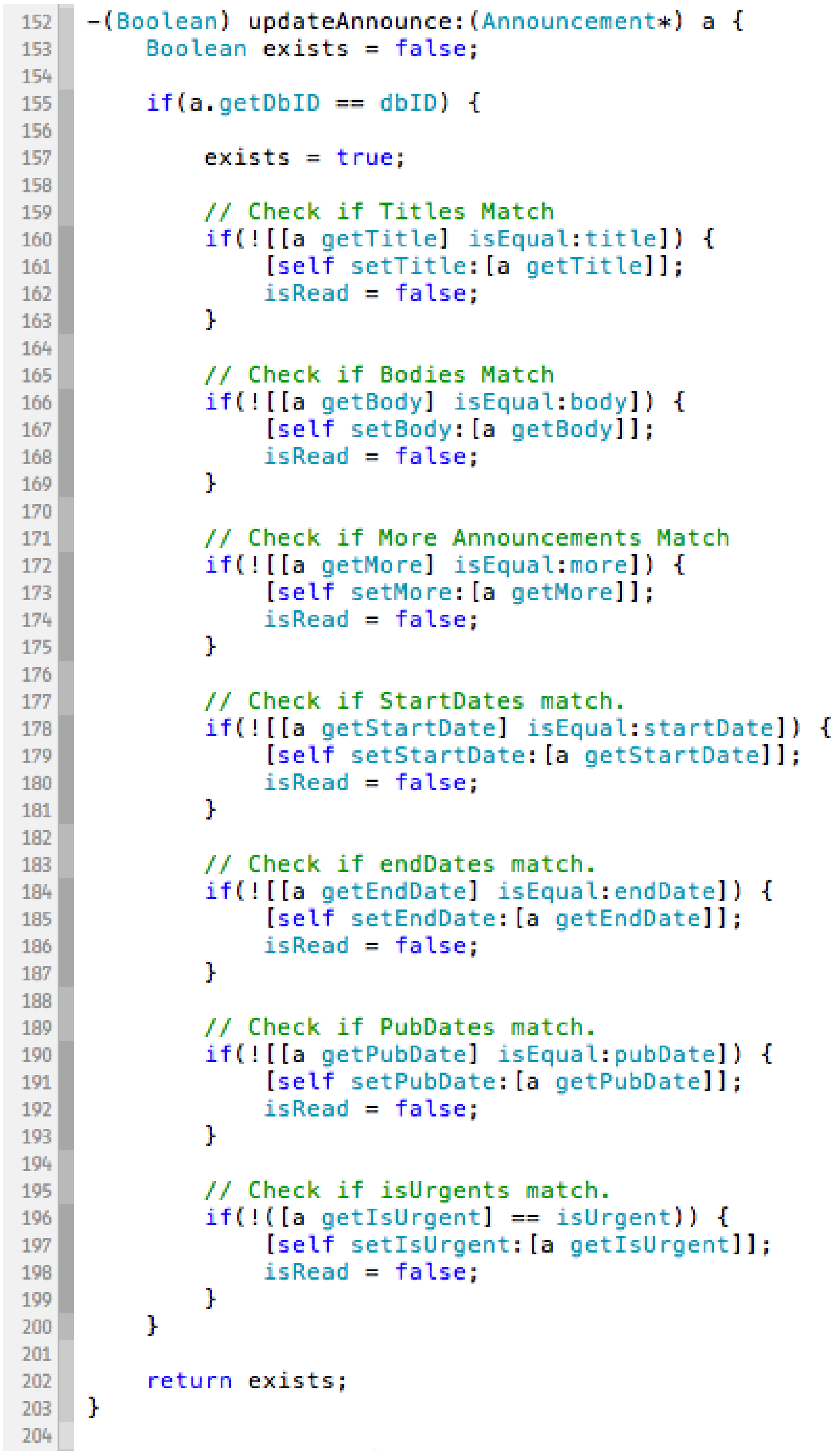
Figure 72. muster iOS Method

Figure 73 and 74 are the updateAnnounce() methods. Each takes an announcement object recently retrieved from the remote database and compares it to itself. This is only necessary due to limitations with the database schema with which we worked.



```java

    public boolean updateAnnounce(Announcement a){
        boolean exists = false;
        if(a.getDbID() == this.dbID){


            //Announcement already exists
            exists = true;
            //If Titles don't match.
            if(!a.getTitle().equals(this.getTitle())){
                this.setTitle(a.getTitle());

                isRead = false;
            }

            //If Bodies don't match.
            if(!a.getBody().equals(this.getBody())){
                this.setBody(a.getBody());
                isRead = false;
            }

            //If More don't match.
            if(!a.getMore().equals(this.getMore())){
                this.setMore(a.getMore());
                isRead = false;
            }

            //If StartDates don't match.
            if(!a.getStartDate().equals(this.getStartDate())){
                this.setStartDate(a.getStartDate());
                isRead = false;
            }

            //If endDates don't match.
            if(!a.getEndDate().equals(this.getEndDate())){
                this.setEndDate(a.getEndDate());
                isRead = false;
            }

            //If PubDates don't match.
            if(!a.getPubDate().equals(this.getPubDate())){
                this.setPubDate(a.getPubDate());
                isRead = false;
            }

            //If isUrgent don't match.
            if(!a.isUrgent == this.isUrgent){
                this.setUrgent(a.isUrgent);
                isRead = false;
            }
        }
        return exists;
    }

```

Figure 73.  updateAnnounce() Android Method



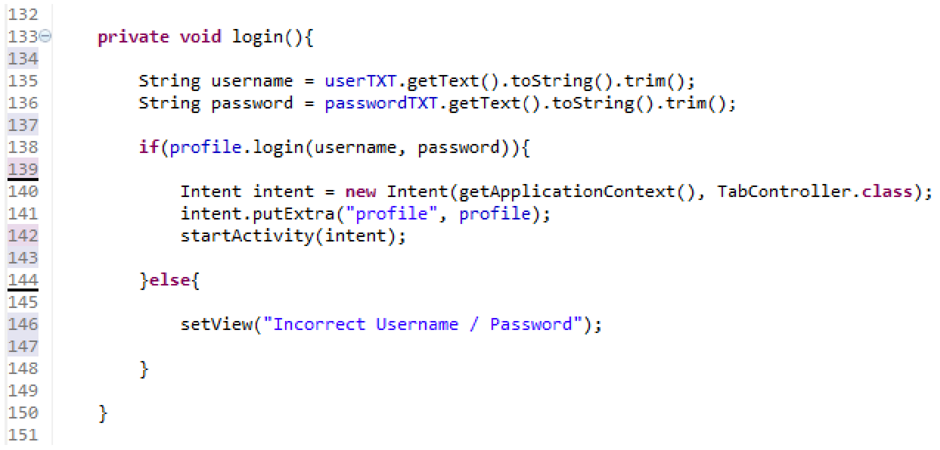
```
-(Boolean) updateAnnounce:(Announcement*) a {
    Boolean exists = false;

    if(a.getDbID == dbID) {

        exists = true;

        // Check if Titles Match
        if(![[a getTitle] isEqual:title]) {
            [self setTitle:[a getTitle]];
            isRead = false;
        }

        // Check if Bodies Match
        if(![[a getBody] isEqual:body]) {
            [self setBody:[a getBody]];
            isRead = false;
        }

        // Check if More Announcements Match
        if(![[a getMore] isEqual:more]) {
            [self setMore:[a getMore]];
            isRead = false;
        }

        // Check if StartDates match.
        if(![[a getStartDate] isEqual:startDate]) {
            [self setStartDate:[a getStartDate]];
            isRead = false;
        }

        // Check if endDates match.
        if(![[a getEndDate] isEqual:endDate]) {
            [self setEndDate:[a getEndDate]];
            isRead = false;
        }

        // Check if PubDates match.
        if(![[a getPubDate] isEqual:pubDate]) {
            [self setPubDate:[a getPubDate]];
            isRead = false;
        }

        // Check if isUrgents match.
        if(!([a getIsUrgent] == isUrgent)) {
            [self setIsUrgent:[a getIsUrgent]];
            isRead = false;
        }
    }

    return exists;
}
```

Figure 74. updateAnnounce iOS Method

## 6. Add Functionality

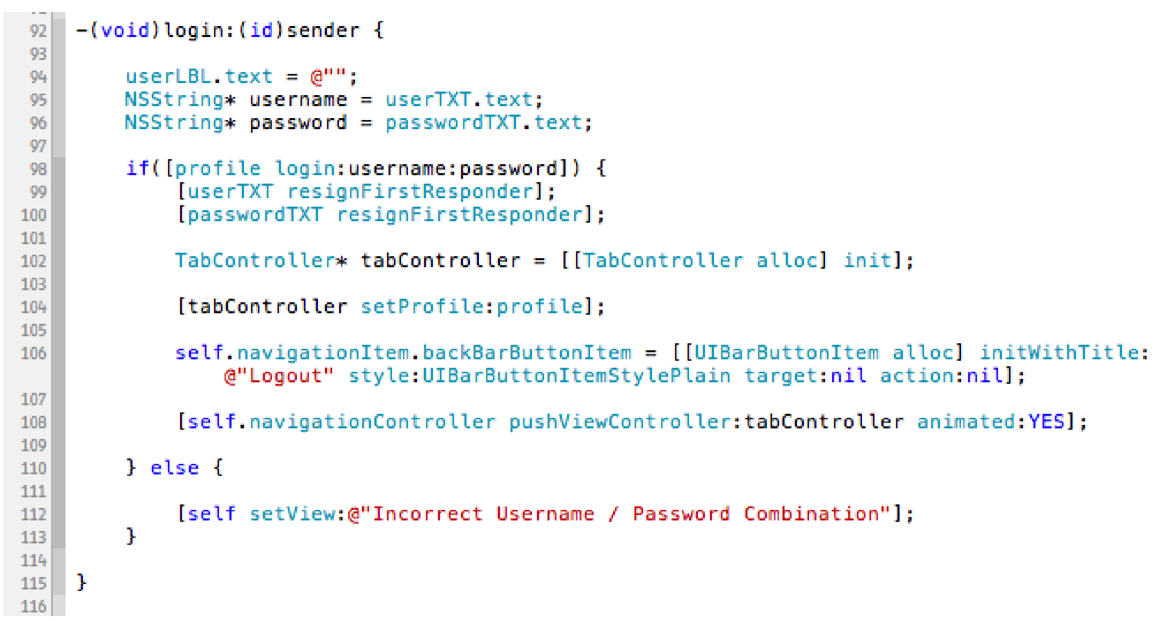
After implementing the models, it is time to tie everything together. The Controller will tie the Models together with the Views. The Controller will respond to user actions as received by View objects and modify the Models as required to produce necessary functionality. At this point, the developer should be able to use the various Models and their methods to complete the desired actions.

In Figures 60 and 68 of the implement navigation section there is a login function that uses the stub-code, if(true). That code was used to test the navigation. Since the Models have been completed the developer can now replace that stub-code with the profile method calls. The Android stub-code can be replaced with if(profile.login(username, password)) as displayed in Figure 75 and the iOS code can be replaced with if([profile login:username:password]) as displayed in Figure 76.



```
    private void login(){

        String username = userTXT.getText().toString().trim();
        String password = passwordTXT.getText().toString().trim();

        if(profile.login(username, password)){

            Intent intent = new Intent(getApplicationContext(), TabController.class);
            intent.putExtra("profile", profile);
            startActivity(intent);

        }else{

            setView("Incorrect Username / Password");

        }

    }
```

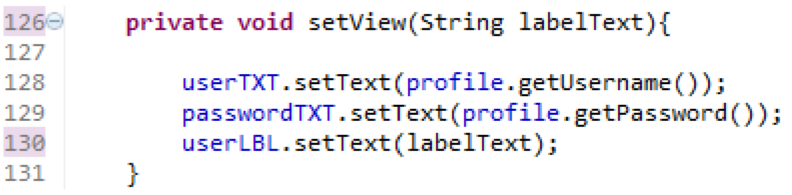
Figure 75. login() Android Method



```
-(void)login:(id)sender {

    userLBL.text = @"";
    NSString* username = userTXT.text;
    NSString* password = passwordTXT.text;

    if([profile login:username:password]) {
        [userTXT resignFirstResponder];
        [passwordTXT resignFirstResponder];

        TabController* tabController = [[TabController alloc] init];

        [tabController setProfile:profile];

        self.navigationItem.backBarButtonItem = [[UIBarButtonItem alloc] initWithTitle:
            @"Logout" style:UIBarButtonItemStylePlain target:nil action:nil];

        [self.navigationController pushViewController:tabController animated:YES];

    } else {

        [self setView:@"Incorrect Username / Password Combination"];
    }

}
```

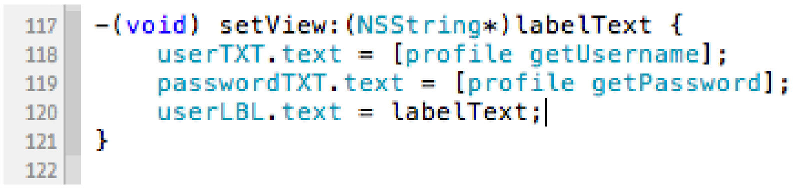
Figure 76.    login iOS Method

We also need to finish our setView() method, Figure 77 and 78 shows these completed methods.



```
    private void setView(String labelText){

        userTXT.setText(profile.getUsername());
        passwordTXT.setText(profile.getPassword());
        userLBL.setText(labelText);
    }
```

Figure 77.    setView() Android Method



```
-(void) setView:(NSString*)labelText {
    userTXT.text = [profile getUsername];
    passwordTXT.text = [profile getPassword];
    userLBL.text = labelText;
}
```

Figure 78.    setView iOS Method

This process should be repeated for each Controller class. After all controllers have been modified and fitted with proper functionality, then it is time to begin testing on each platform, followed by deployment.

## G. CONCLUSION

The design process we have described will aid developers in minimizing the design process when creating applications for multiple platforms. This process in no way seeks to replace a solid understanding of either platform; rather, it seeks to provide a common approach with which a developer may build and maintain applications across multiple platforms. Even with language differences and some implementation differences, the applications should align to at least the screen level, if not to the method level. Once the application is built on either platform, the developer should have much less difficulty maintaining a consistent set of features between the platforms. Further, such consistency is beneficial to the maintenance of the applications following deployment; it is well established within the software engineering domain that the bulk of a software project’s expense is associated with the maintenance or post-development phase.

# VII. SUMMARY AND CONCLUSION

Cross-platform mobile application development can benefit from a common design process. Since such development needs to span language barriers, developers cannot share code modules between platforms. Therefore, we introduced common design concepts, such as Screens and Stack, Horizontal, and Composite Navigation. We also showed how to apply the Model-View-Controller (MVC) design pattern to the Android platform in order to apply a Unified Design Process.

Functionality that is common across all platforms should be replicated in Model Logic and written as similarly as possible, despite language differences. By using the MVC pattern in the early design phase, a majority of the application logic can be pushed into Model objects. These Models are simple to implement and only require the developer to do simple language translations for each platform. The Controller classes in our projects became the bridge between the screens and the Models. By eliminating the interaction between Models and Views we were able to separate out the UI design phase and ensure that each platform was designed according to separate UI Guidelines. This provides users with a platform specific look-and-feel while maintaining similar functionality between platforms.

We also identified functional areas that help developers determine which sections of the controllers can be directly translated. However, due to API differences there are minor issues with direct translations. Our research also found that due to the nature of the Android back stack and its inability to branch, the implementations of composite navigation would differ slightly.

Each application project, while different, has many common concepts and patterns. The application of these common ideas in our Unified Design Process will reduce the amount of time that is spent in the design phase of application development.

## A. FUTURE WORK

Our design process has been tested only on numerous simple applications with limited feature set and a capstone project that incorporated more advanced functionality combinations. While the process provides promising results, there still remains a substantial amount of work to do before it can be widely acceptable. The following sections discuss future areas that still require focus.

### 1. Security

Little effort was made during our development process to implement solid security measures. We relied on the secure nature of the platforms we were using to implement our designs. Security in any application, let alone one that is intended to connect to DoD systems, must include a robust security model. Future research should focus on security patterns that may be implemented in mobile applications, regardless of targeted platform.

### 2. Synchronizing Announcements

We introduced the concept of synchronizing announcements to a mobile device from a remote server. While we did implement an algorithm capable of managing this process, we find that it is far from efficient. We suggest that future researchers look into the possible use of a hashing function for announcements or some other form of announcement comparison. We believe that patterns may be developed to encompass such actions on a more abstract layer, for example a synchronization pattern may be developed that defines how a developer may synchronize two data elements between a remote location and a local device.

### 3. Additional Platforms

We limited the scope of this thesis to two platforms, iOS and Android. It would be interesting to explore the viability of applying this design process to additional platforms, such as Windows Phone and Symbian. We are confident that the design process is applicable to any mobile platform or highly UI intensive cross-platform application.

### 4. System Login

During the actual development of the NPS Muster application, we were unable to gain access to a system that would allow for the verification of user credentials. We currently simulate this feature with test methods that return either true or false values. If the NPS Muster application is to be deployable it must have a way to validate user credentials with the current user directory and network access control schema in use at the Naval Postgraduate School.

### 5. Web Services

We also considered the use of web services as a means of cross-platform programming. Hosting services on a server would require a developer to create a service and an interface for each platform. There are certain limitations to this concept, such as application functionality in offline environments, as well additional security concerns. If those issues were to be controlled in some manner, offloading certain application logic to web services would reduce code duplication and simplify maintainability.

### 6. Cross-Platform Programming with Web Applications

Another viable solution to the cross-platform programming domain is the use of Web Applications. With the current shift of the Internet to HTML5, the mobile device’s web browser becomes a portal into the world of applications. We conducted some initial research on the use of Web Applications to accomplish the “design-once” philosophy, but decided we did not want to use a “neutral” UI, one that is exactly the same on all devices and does not adhere to any specific UI guidelines. Our intent was to focus on maintaining the platform specific UIs and their associated “look-and-feel.”

### 7. Cross-Platform Programming with OpenGL

Similar to the concept of using Web Applications that are applicable regardless of platform, we envisioned the use of OpenGL to facilitate another “design-once” concept. OpenGL is supported on both Android and iOS. Therefore, it is possible that if an application is designed and built using only OpenGL constructs it can be run on both platforms.

THIS PAGE INTENTIONALLY LEFT BLANK

# LIST OF REFERENCES

[1] Google, Inc. *Platform versions*. [Online]. Available: http://developer.android.com/resources/dashboard/platform-versions.html

[2] Google, Inc. *What is Android?* [Online]. Available: http://developer.android.com/guide/basics/what-is-android.html

[3] M. Cleron. (2007). *Androidology - Architecture overview.* [Video File]. Available: http://www.youtube.com/watch?v=Mm6Ju0xhUW8

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wessley Longman Publishing Co, Inc, 1995.

[5] S. Burbek. *Application Programming in Smalltalk-80: How to Use Model-View-Controller (MVC), 1997* [Online]. Available: http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html

[6] Google, Inc. *View.* [Online]. Available: http://developer.android.com/reference/android/view/View.html, [Nov. 14, 2011].

[7] Apple, Inc. *UIView Class Reference*, 2001. [Online]. Available: https://developer.apple.com/library/ios/#documentation/UIKit/Reference/UIView_Class/UIView/UIView.html

[8] Google, Inc. *Activities*. [Online]. Available: http://developer.android.com/guide/topics/fundamentals/activities.html

[9] Apple, Inc. *About View Controllers.* [Online]. Available: https://developer.apple.com/library/ios/#featuredarticles/ViewControllerPGforiPhoneOS/Introduction/Introduction.html

[10] Apple, Inc. *Tab Bar Controllers.* [Online]. Available: https://developer.apple.com/library/ios/#documentation/WindowsViews/Conceptual/ViewControllerCatalog/Chapters/TabBarControllers.html#//apple_ref/doc/uid/TP40011313-CH3-SW1

[11] Apple, Inc. *Combined View Controller Interfaces.* [Online]. Available: https://developer.apple.com/library/ios/#documentation/WindowsViews/Conceptual/ViewControllerPGforiOSLegacy/CombiningViewControllers/CombiningViewControllers.html#//apple_ref/doc/uid/TP40011381-CH104-SW1

[12] Apple, Inc. *Model-View-Controller.* [Online]. Available: https://developer.apple.com/library/ios/#documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html

[13] Apple, Inc. *UIWindow Class Reference.* [Online]. Available: https://developer.apple.com/library/ios/#documentation/UIKit/Reference/UIWindow_Class/UIWindowClassReference/UIWindowClassReference.html

[14] Apple, Inc. *App States and Multitasking.* [Online]. Available: https://developer.apple.com/library/ios/#documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/ManagingYourApplicationsFlow/ManagingYourApplicationsFlow.html

[15] Apple, Inc. *UIApplication Class Reference.* [Online]. Available: https://developer.apple.com/library/ios/#documentation/UIKit/Reference/UIApplication_Class/Reference/Reference.html

[16] Apple, Inc. *The View Controller Lifecycle.* [Online]. Available: https://developer.apple.com/library/ios/#featuredarticles/ViewControllerPGforiPhoneOS/ViewLoadingandUnloading/ViewLoadingandUnloading.html

[17] Google, Inc. *Tasks and Back Stacks.* [Online]. Available: http://developer.android.com/guide/topics/fundamentals/tasks-and-back-stack.html

[18] Google, Inc. *Activity*. [Online]. Available: http://developer.android.com/reference/android/app/Activity.html

THIS PAGE INTENTIONALLY LEFT BLANK

## INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California